GAME DEVELOPER BOOKS
Reverse Dictionary of Game Programming

逆引き ゲームプログラミング

Reverse Dictionary of Game Programming

著 万里 ゆうじ

for Windows DirectX

秀和システム

TECHNICAL MASTER はじめてのCプログラミング改訂版 C言語ステップアップ編

著　者：砂田紀一郎
定　価：（本体2600円＋税）
ISBNコード：4-7980-1036-7
2005/03/29　B5変　512頁　CD付き

本書は、C言語の初歩をマスターした人が、さらにステップアップして、より実践的なプログラムを組むための入門書です。ポインタや構造体、メモリ操作やアルゴリズムなど、実際にプログラムを組む上で必要な知識やテクニックをやさしく解説しました。姉妹編の「C言語基本マスター編」とあわせてお読みください。CD-ROMにはBorland C++ Compiler5.5、C++ BuilderX Personalとサンプルプログラムを収録しました。

Visual C++ .NET 逆引き大全 500の極意

著　者：岩田宗之
定　価：（本体2500円＋税）
ISBNコード：4-7980-1096-0
2005/06/22　A5　718頁

本書はマイクロソフト社の.NET Frameworkに対応した「VisualC++.NET」の解説書です。.NET開発環境下での操作の実際を、C言語初心者には基礎的な文法から中級者にはC++の新たなライブラリとWindowsプログラミングを中心に、自分のレベルに合わせて利用可能な、現場で役立つ500のテクニックを紹介します。サンプルデータダウンロードサービス付き。

VisualC#.NET 逆引き大全 500の極意

著　者：池谷京子、増田智明
定　価：（本体2600円＋税）
ISBNコード：4-7980-0813-3
2004/07/24　A5　656頁　CD付き

Microsoft社のプログラミング言語、Visual C# .NETの基本テクニックから、開発ですぐに役に立つテクニック、一歩進んだテクニックが満載の1冊です。サンプルプログラムとともに解説するテクニック数は合計500項目にのぼり、コードの逆引き事典としても使えます。 また、添付CD-ROMには、本文中で紹介したサンプルプログラムが収録されています。

COVER DESIGN：YAMANA (linkup)

GAME DEVELOPER BOOKS
Reverse Dictionary of Game Programming

逆引き ゲームプログラミング

Reverse Dictionary of Game Programming

著 万里 ゆうじ

秀和システム

■注意

1.本書は著者が独自に調査した結果を出版したものです。

2.本書は内容において万全を期して制作しましたが、万一不備な点や誤り、記載漏れなどお気づきの点がございましたら、出版元まで書面にてご連絡ください。

3.本書の内容の運用による結果の影響につきましては、上記2項にかかわらず責任を負いかねます。あらかじめご了承ください。

4.本書の全部または一部について、出版元から文書による許諾を得ずに複製することは禁じられています。

5.商標について

・Microsoft ならびに Windows、DirectX は、米国 Microsoft Corporation の米国およびその他の国における商標または登録商標です。

・Borland、ならびに Borland C++ Compiler は米国 Borland 社の商標または登録商標です。

その他、本書に登場するシステム名称、製品名は一般に各社の商標または登録商標です。なお、本文中には ™、® マークを明記しておりません。

― はじめに ―

本書を手にした方は、ゲーム製作に興味のある方だと思います。

しかしゲーム製作の方法は多岐に渡るため、全てのジャンルの制作方法を紹介する事はなかなか困難です。

そのため本書では、多様なサンプルを載せて、それらを解説する形を取りました。

もし使えそうな処理がありましたら、読むだけではなく、是非読者自身の手で、改良を施してみて下さい。

著者

Contents 目次

Chapter 1 導入編 1

Chapter 2 並列動作 7

Chapter 3 システム 31

Chapter 5　グラフィックの表示　173

Chapter 6　移動　241

Chapter 8 当たり判定 387

Chapter 9 アイテム処理 453

Chapter 10 ゲームバランス 493

Chapter 11　サウンド　515

Chapter 12　その他　547

Chapter 13　TIPS　587

本書の使いかた

読み方

基本的に、頭から順番に読む必要はありません。

各章は、処理内容のジャンルごとに分類されていますので、それぞれを組み合わせて使ったり、ふと知りたくなったところを目次から探して、辞書のように使ったりするなど、自由にお使いください。

サンプルリスト

サンプルリストはソースコードを全て掲載している訳ではありません。解説に必要な所のみとなっています。

最初から最後までのソースコードを見たい場合は、秀和システムの本書Webページより、サンプルプログラムをダウンロードしてください※。

ページレイアウト

以下のように、ソースの一行が長い場合、ページレイアウトの都合上、次の行に改行されています。しかし、ソースコードの上では、下線で区切られていない限り改行はしていませんので、紙面を参考に手で入力する場合などは、注意してください。

また、紙面上には解説している部分のソースのみ掲載している場合があります。全てのソースコードを参照したい場合は、ダウンロードしたサンプルをご覧下さい。

▼改行の解説例

この行はソースコード上は改行していない

```
//同一座標の場合エラーが出るので特別処理
    if( (work->Target.X == work->MyShip.X )&&(work->Target.Y == work-
>MyShip.Y) )
    {
        distance = 0;
    }else{
        //相手との距離を計測
        distance = sqrtf( (work->Target.X - work->MyShip.X) * (work-
>Target.X - work->MyShip.X) +
```

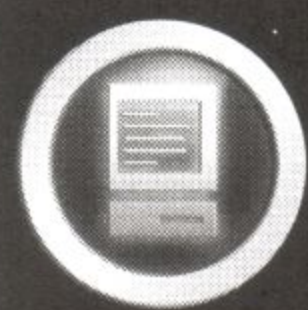
この行はソースコード上は改行していない

※ダウンロードに関しては、1章参照。

Chapter 1

導入編

逆引き ゲームプログラミング

Game Programming

1-1 DirectXのインストール

DirectX

DirectXとは、Windows上でゲームを作成するために用意された一連のAPIの事です。

本書ではこのDirectXを使用してプログラムの作成、解説を行なっていきます。

そのため、サンプルのプログラムを作成、実行するために、DirectXに対応したハードウェア環境と、最新のDirectXSDKが必要になります。

ハードウェアについて

実行するハードウェアの環境ですが、使用するPCが以下の条件に対応している事を確認してください。

Intel Pentium3 以上のCPU
DirectX8.1に対応し、VRAMを32M以上搭載するビデオカード
DirectX8.1に対応したサウンドカード

SDKについて

SDKとは、Software Develop Kit（ソフトウェア開発キット）の略で、このSDKをインストールする事によって、DirectXを使用したアプリケーションの開発が可能になります。

以下の手順にそって、インストールを行なってください。なお、2005年10月現在の最新版は「Microsoft DirectX 9.0 SDK Update (October 2005)」です。

SDKのインストール方法

まず、マイクロソフトのウェブサイトから、最新のDirectXSDKを入手します。

サイトのURLアドレスは、以下の通りです。

http://www.microsoft.com/japan/msdn/directx/downloads.aspx

ここから、「ソフトウェア開発キット」(Microsoft DirectX 9.0 SDK Update (October 2005))をダウンロードして下さい※。

ダウンロード後、入手した実行ファイル「dxsdk_aug2005.exe」を実行します。

あとは、表示される指示にしたがって、インストールを行なってください。

※2005年10月時点の最新。

1-2 コンパイラのインストール

使用するコンパイラ

本書ではプログラムの作成にBorland社のコンパイラである、Borland C++ Compiler 5.5（以下BCC）を使用しています。

そのため、サンプルプログラムをコンパイルするには、BCCのインストールが必要です。

このコンパイラは無償で配付されており、Borland社のホームページからダウンロードする事が出来ます※。

下記にインストールするための手順を示します。

・1　BCCをボーランド社のホームページからダウンロードします。ダウンロードサイトは以下のURLです。

http://www.borland.co.jp/cppbuilder/freecompiler/bcc55steps.html

書かれている内容に従って、BCCをダウンロードして下さい。

・2　ダウンロードしたファイルを実行します。

・3　インストールするパスを指定し、そのまま完了を押します。

以上で、インストールは終了です。

なお、デフォルト以外のパスにインストールした場合、サンプルプログラムの設定は自分で行なう必要があります。

※ダウンロードには「ボーランド MyPage」のメンバー登録が必要です。

1-3 サンプルプログラムのコンパイルと実行

インストール

DirectX とコンパイラのインストールがすんだら、サンプルプログラムをコンパイルしてみましょう。

ただしその前に、サンプルプログラムの入手と、インストールを行なう必要があります。

以下の手順でインストールを行なってください。

・1　**サンプルプログラムを秀和システムのサイトから入手します。**

トップページで本書名で検索をするか、以下のURL を直接入力してください。
そのページの【追加情報】のサポート情報を見るをクリックしてください。

http://www.shuwasystem.co.jp/cgi-bin/detail.cgi?isbn=4-7980-1169-X

ダウンロードのページに飛びますので、そのページから、サンプルファイル（ファイル名：SAMPLE.exe）をダウンロードしてください。

・2　**ダウンロード終了後、ファイル（SAMPLE.exe）を実行して下さい。**

インストールするフォルダを聞いてきますので、HDD 内の任意のフォルダを指定します。
その後は、画面の指示に従ってください。

・3　**終了後、インストールしたフォルダ内にある、"SET_FILE.BAT"を実行します**※。

以上でインストールは終了、コンパイルの準備が整いました。

※ BCC をデフォルト以外のフォルダにインストールした場合、このSET_FILE.BAT をフォルダに合わせて変更する必要があります。

サンプルプログラムのコンパイル

それでは実際にサンプルプログラムをコンパイルしてみます。

インストールしたフォルダ内の"¥SAMPLE"フォルダ内にある、"SAMPLE.BAT"を実行してください。

すると、コンパイル環境が設定されたコマンドラインプロンプトが開きます。

ここで、make と打ち込んでリターンキーを押してください。

サンプルプログラムのコンパイルが行なわれていきます。

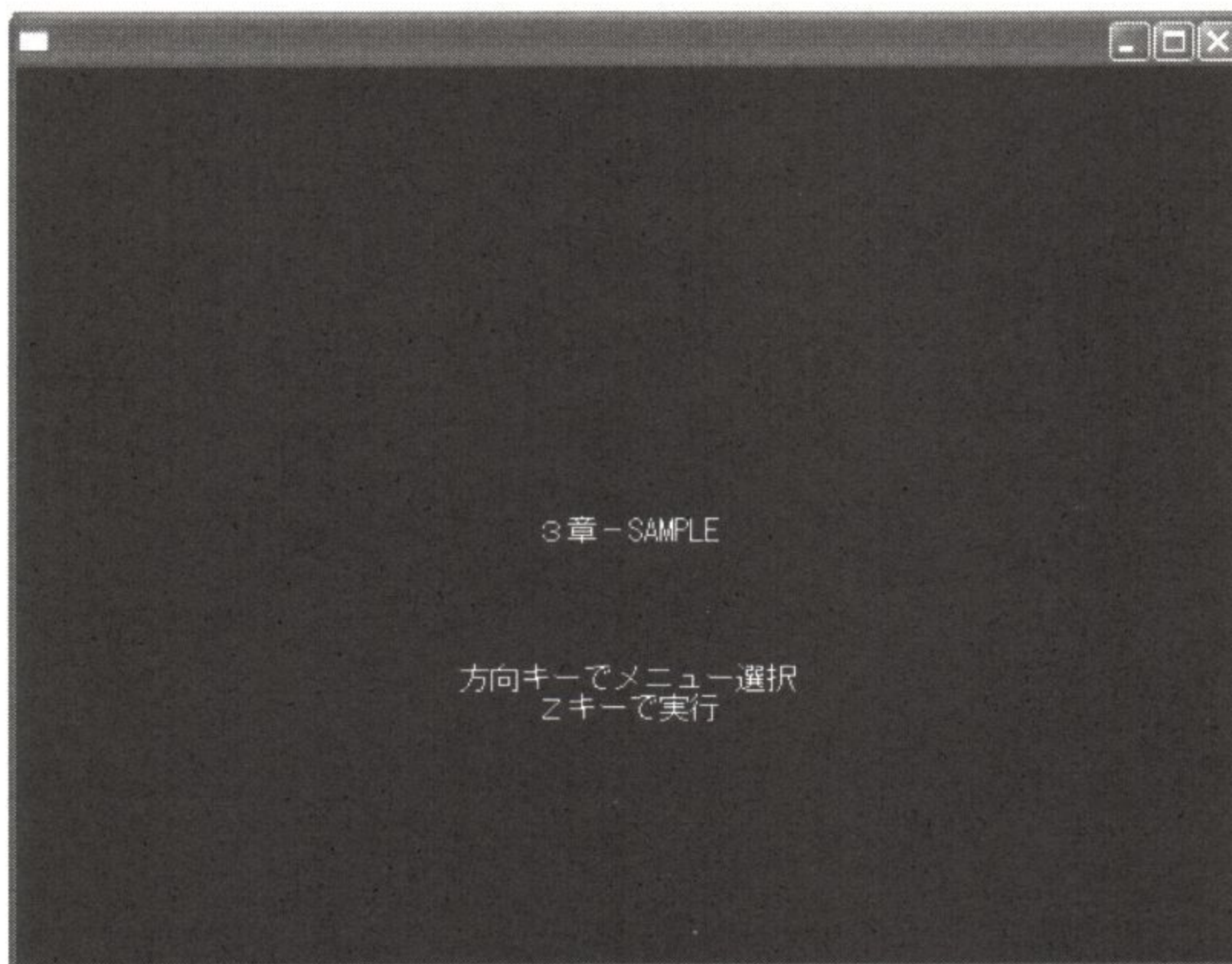
フォルダ「EXEC」の中に、SAMPLE.EXE が出来ていれば成功です。

サンプルプログラムの実行

サンプルプログラムを立ち上げると、実行するサンプルの選択画面が現れます。

カーソルキーで実行するサンプルを選択し、Z キーで実行します。

図1 ー3ー1 実行画面

Chapter 1

1-4 DirectX 用のライブラリ

ライブラリの構築

本来、Borland C++ Compiler 5.5 は、そのままでは DirectX を利用する事ができません。
使用する為には、使用する DirectX に対応したライブラリを用意してやる必要があります。
このライブラリはサンプル内にすでに含まれていますが、自分で構築する事もできます※。
その場合は、下記の手順に従って作成して下さい。

・1　DirectX と、Borland C++ Compiler 5.5 がきちんとインストールされているか確認します。

・2　サンプルをインストールしたフォルダ内の "¥DEVELOP¥MAKE_LIB¥DLL" フォルダに、Windows のシステムフォルダ "C:¥WINDOWS¥system32" から、以下のライブラリファイルをコピーします。

d3d9.dll
d3d9d.dll
d3dx9_27.dll
d3dx9d_27.dll
dsound.dll
dmusic.dll

・3　"¥DEVELOP¥MAKE_LIB" フォルダ内にある "makelib.bat" を実行します。

以上の手順で、ライブラリが作成できます。
作成したライブラリはプログラムからリンクできるフォルダに移動するのを、忘れないようにして下さい。

※[1 ー 3]のインストール手順の3番目でライブラリの設定とコピーを行なっています。

Chapter 2

並列動作

2-1 ゲームはどうやって動いているのか？

ゲームプログラミングは特殊？

ゲームというのは、簡単に操作できて取っ付きが良い反面、どうやって作成されているのか良く分からない、という意見を耳にします。

これは、ジョイスティック等のWindowsでは余り一般的ではない入力処理に加え、画面を動き回る色々な物体の移動等、いわゆる「ゲームでしか見られない処理」が多いため、そういったイメージが先行しているのだと思います。

もちろん、ゲームとはいえコンピュータ上で動いている以上、プログラムによって動いている事は言うまでもありません。

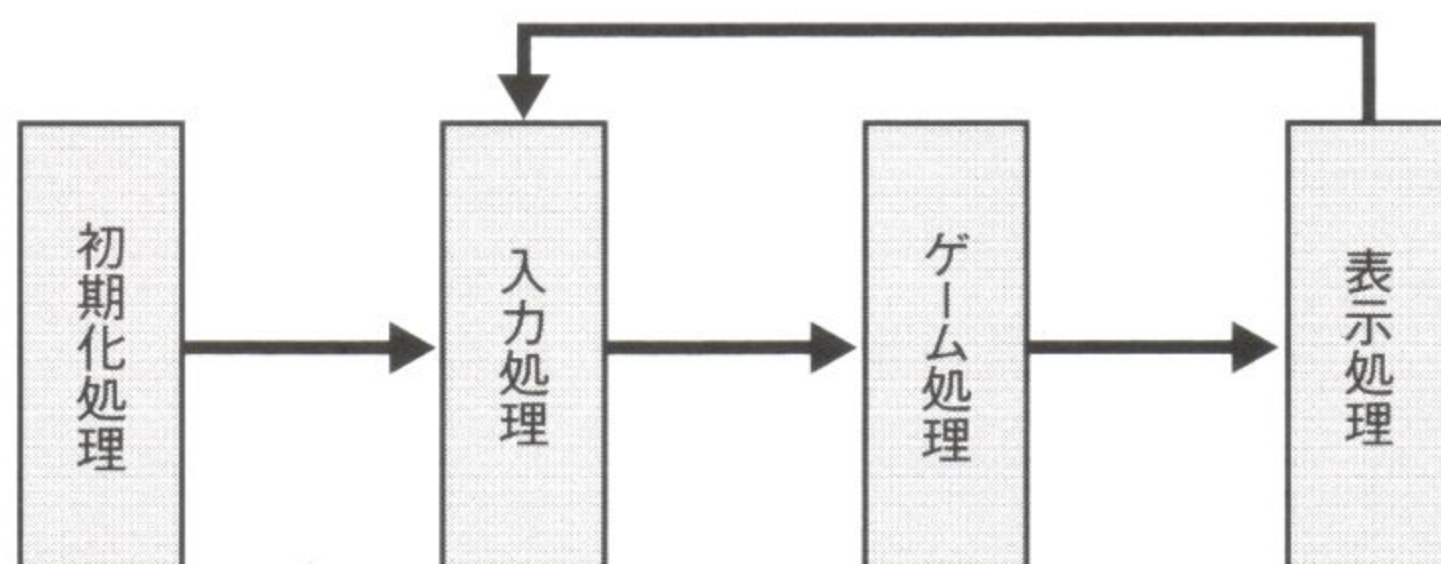
ここで問題となるのは、「どういう仕組みで」ゲームが動いているかという事でしょう。

基本的には普通のプログラムとなんら変わるところはありません。ですが、ゲームならではの処理、というのはあります。

例えばゲームでは、多数のキャラクターを同時に扱うケースが多く、そういった事を扱いやすくするために簡易的な並列動作のシステムを搭載する事があります。

他にも、画像の取り扱いが非常に多いので、そういった部分を扱いやすくする工夫も随所にあります。

図2－1－1 ゲームプログラムのイメージフロー

初期化処理 → 入力処理 → ゲーム処理 → 表示処理（→ 入力処理へ戻る）

そういったことを除けば、ゲームプログラムでも、普通のプログラムと同じテクニックやアルゴリズムが使えます。というよりも、本当はそちらのほうがメインです。

上記のようなゲームならではの特殊なプログラムは、ゲームを普通のプログラムのように「楽に」「楽しく」扱うためのテクニックと言えるでしょう。

2-2 実際のゲームの処理の流れとは？

ゲーム処理の流れ

では、実際にゲーム処理の流れとはどのような物でしょうか？
基本的には以下のようになります。

1. プレイヤーがゲームに対して入力をする
2. 入力を元にゲームの処理
3. 処理を元に画像の表示や音声の再生
4. 1に戻る

基本的に、ほぼ全てのゲームはこの流れで動いています。流れとしては非常に単純です。

入力処理の種類

ただ問題となるのは1のプレイヤー入力の部分で、この部分の扱いは大きく2つに分けられます。
入力されるたびに、処理を行なう「イベント方式」と、一定時間ごとに入力をチェックする「処理ループ方式」です。

前者のイベント方式は、Windowsのプログラムでもよく使われる方法で、なにか入力があったら、その瞬間に対応した処理を行ないます。
後者の処理ループ方式は、一定時間ごとに入力をチェックし、それに合わせて処理をします。入力が無い場合は「時間を進めて」1の処理に戻ります。

ゲームのジャンルによって、双方とも向き不向きがありますが、リアルタイムのゲームを扱う場合は、常に時間を更新している処理ループ方式の方が相性が良いようです。
本書でのプログラムは、構成が単純で手軽な事もあり、処理ループ形式を採用しています。

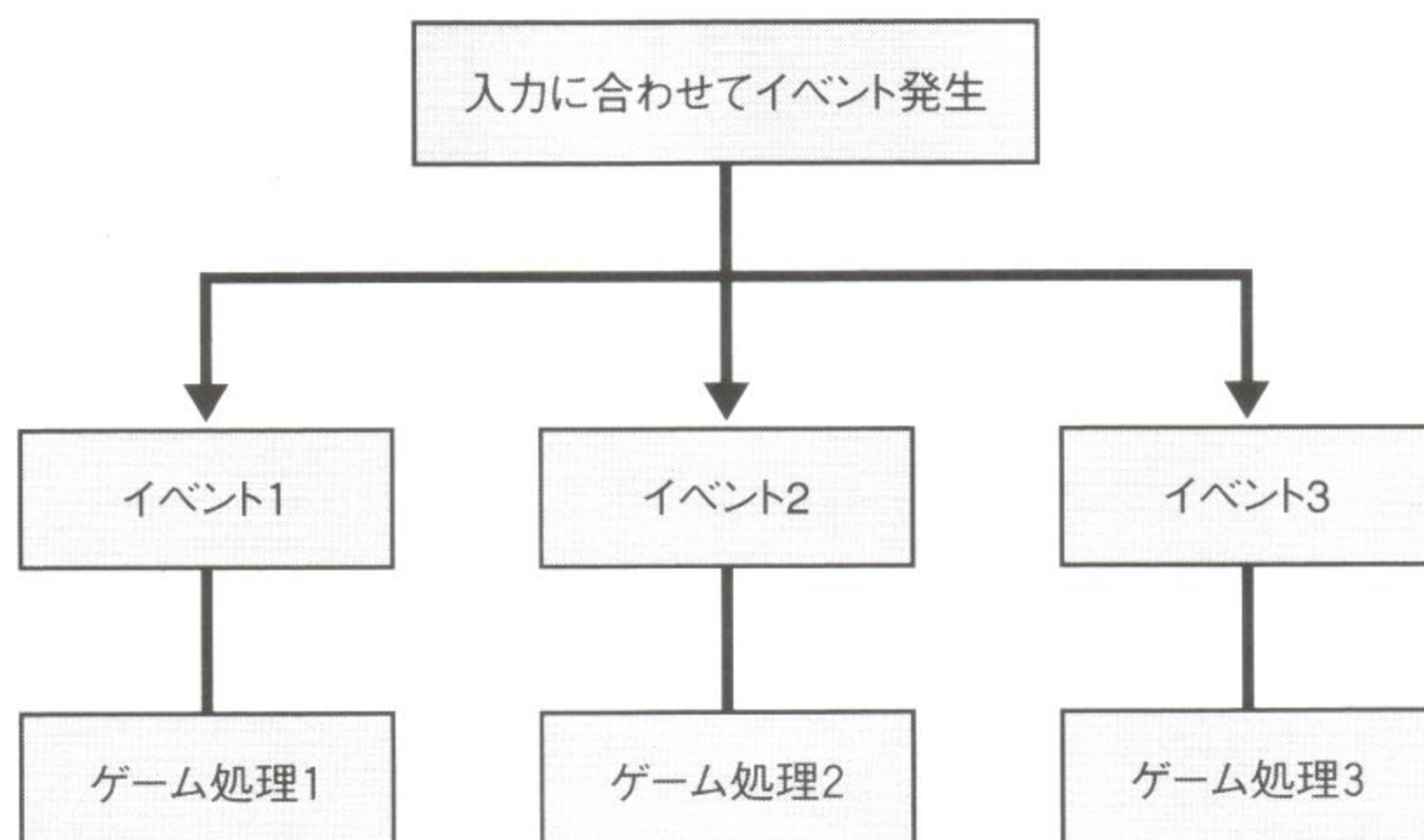
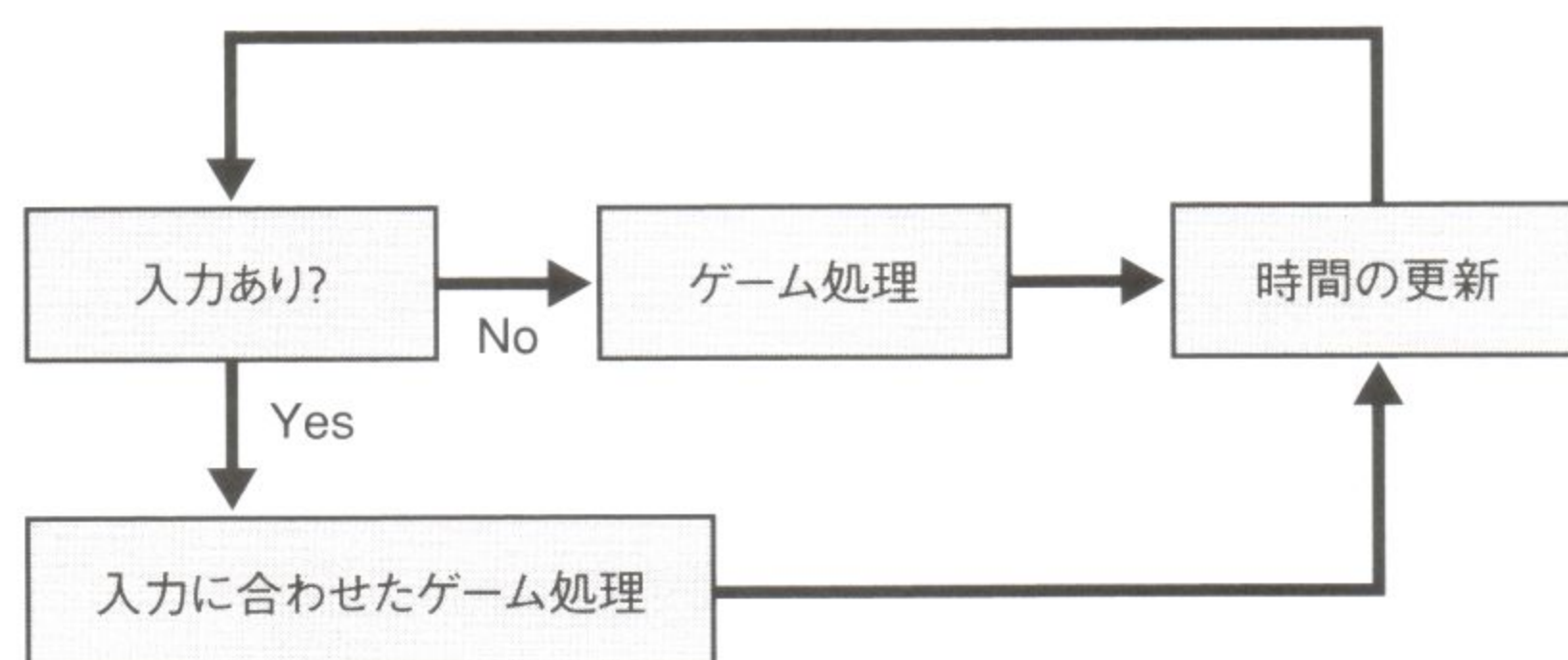
図2－2－1 イベント方式の処理フロー図

入力に合わせてイベント発生

イベント1
イベント2
イベント3

ゲーム処理1
ゲーム処理2
ゲーム処理3

図2－2－2 処理ループ方式の処理フロー図

入力あり?
No
ゲーム処理
時間の更新
Yes
入力に合わせたゲーム処理

2-3 画像の表示はどうやっているのか？

ゲームと画像処理

ゲームは画像処理がその大半を占めているといっても過言ではありません。

ゲーム機等でも、次世代機が発表されるたびに、PR されるのはグラフィック性能が主です。

画像の表示方法は、そのハードウェアによって変わりますが、その扱い方や概念には定石や共通する概念があります。

DirectX での画面の表示

ここでは、その概念を、DirectX を元に解説していきます。

まず下図を見てください。

図2 －3－1 フロントバッファ、バックバッファと、表示切替の概念図

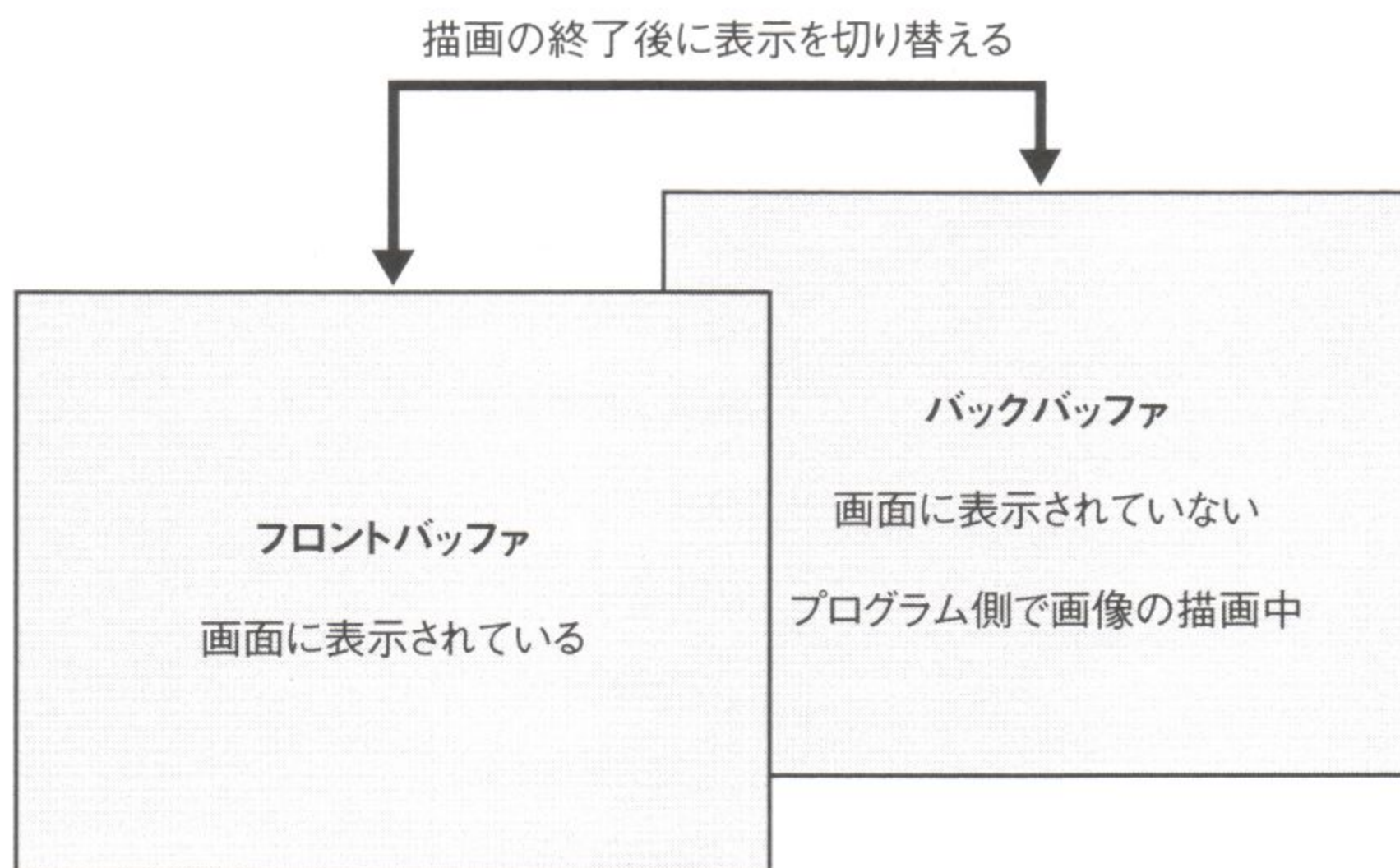
描画の終了後に表示を切り替える

バックバッファ

画面に表示されていない

プログラム側で画像の描画中

フロントバッファ

画面に表示されている

これは、DirectX 上での表示の概念図です。フロントバッファとは、平たく言うと「目に見えている画面」の事です。

同様にバックバッファとは「現在プログラムで処理中の画面」の事です※。プログラムで処理をしているので、実際には見ることはできません。またその性質から、オフスクリーンと呼ばれる事もあります。

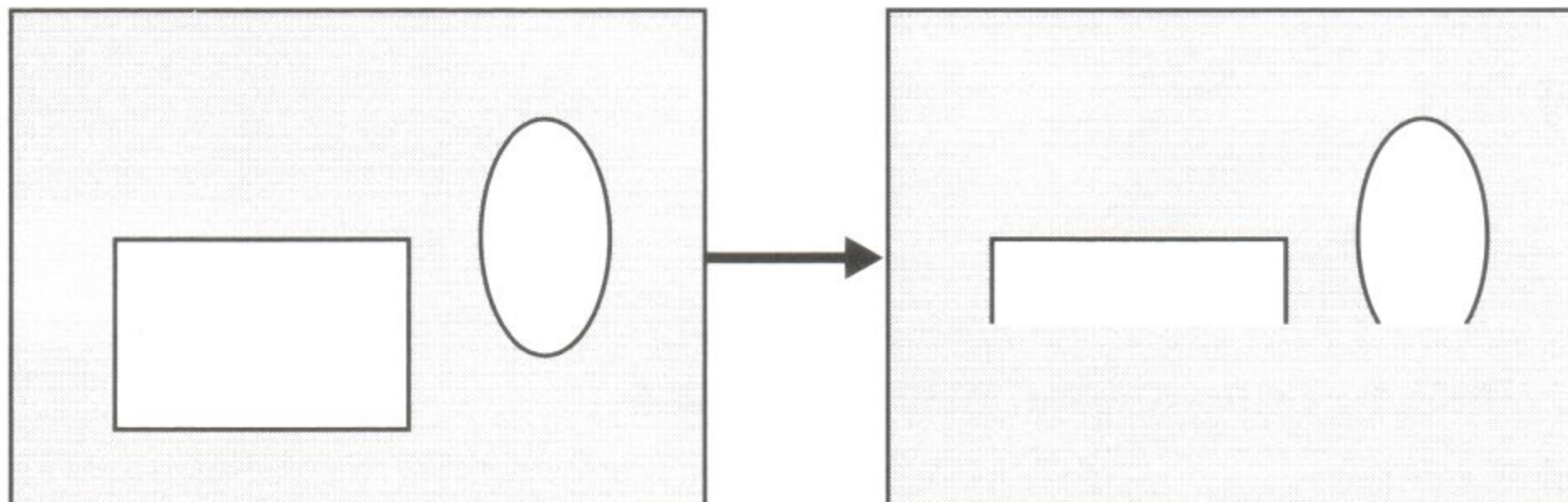
※厳密には少し違うがこう捉えると理解しやすい。

２つに分ける理由

それにしても、なぜわざわざ画面を２つに分けて、このような処理をしているのでしょうか？

理由は非常に単純で、表示中に画像を描き換えると描き換え途中の処理が非常に目立ってしまい、とても見苦しいものになるからです。

図２－３－２ 表示中に画像を描き換えている、イメージ図

本来の画像が完成する前に見えてしまう事があるため、見苦しくなる

2-4 画像の更新はいつ行なっているのか？

画面の更新

プログラムによって、描き終えたバックバッファの画像は、プレイヤーに見せるために、フロントバッファへと切り替えられます。

これを画像(画面)の更新と言います。

処理ループ方式のゲームでは、一定時間ごとにこの画面の更新を行ないます。

そして、この更新の単位を「フレーム」と呼びます。

例えば60フレーム/秒とは、1秒間に60回、画面を更新する事を言います。また、その時間内での処理及び表示を行なっている画像バッファを表す意味もあります。

画面更新回数の決め方

さて、家庭用ゲーム機等では、TV画面の表示周期(リフレッシュレート)に合わせて、約60フレーム/秒で画像の更新をするゲームが大半を占めています※。

これは、画像の更新と時間の更新を同一視すると、表示と時間と処理の同期が完全に取れ、非常にゲームを作りやすくなるためです。

実際この手法は、汎用性は低くなるものの、処理速度が大きく変化しない限り大きな問題は無く、非常にシンプルで優れた手法です。

しかし、Windowsでこの手法を行なうと、画面のモードによって切り替えるタイミングが変わってしまいます。

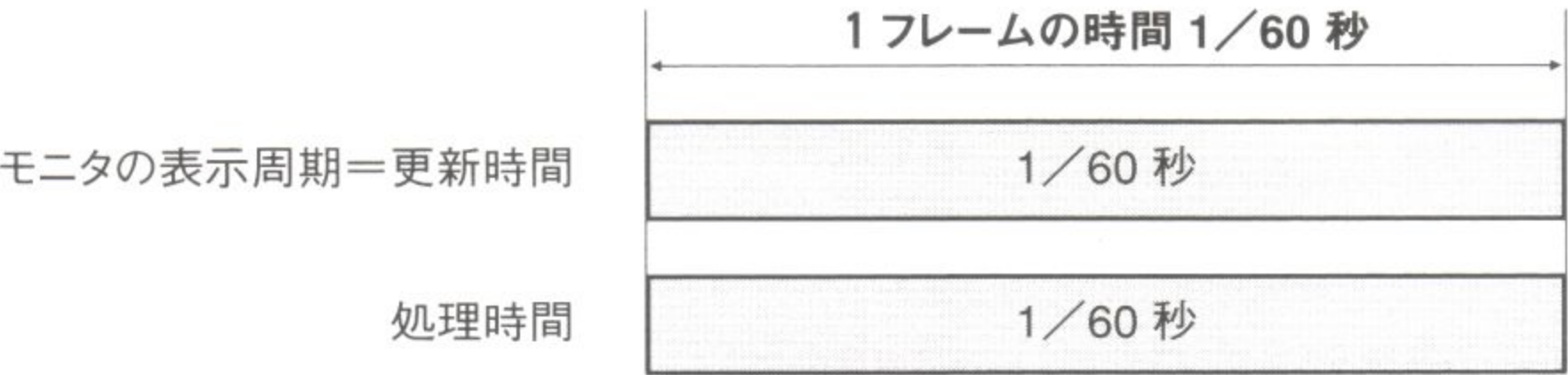
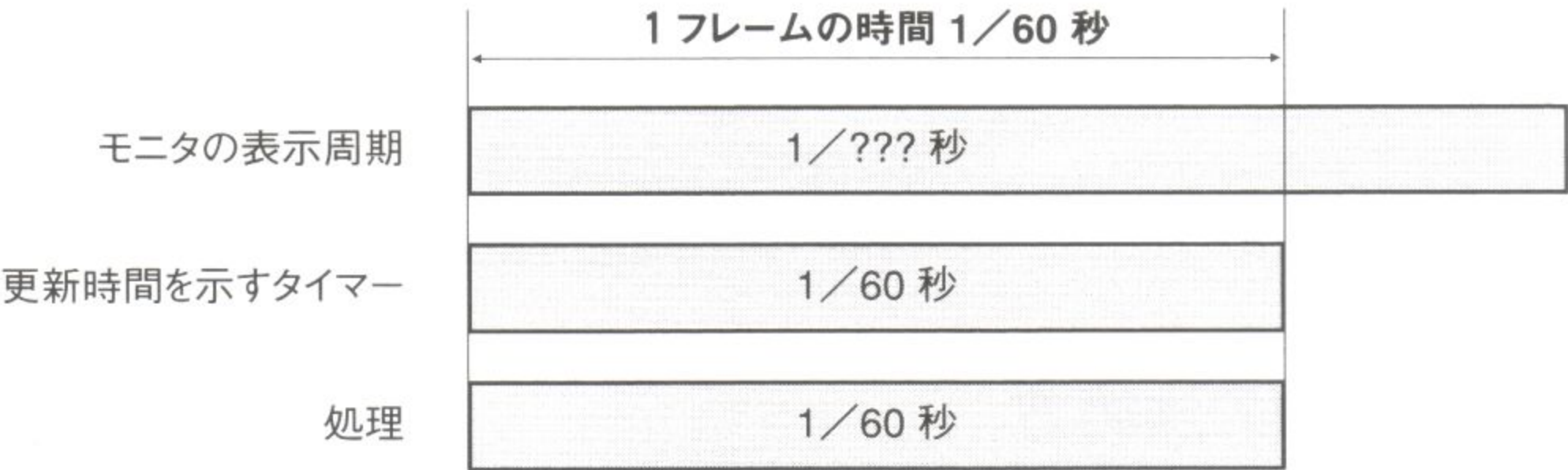
具体的には、1秒間に更新する回数が変わってしまい、環境によってゲームの速度が変わってしまう恐れがあります。

そのため、Windowsでの表示の更新処理では表示とは無関係なタイマーを別途用意し、その時間に合わせて表示画像の更新を行なうのが一般的です。

処理と時間を切り離すための工夫もありますが、若干複雑な処理をはらむため、本書でもこの手法でプログラミングを行なっています。

※正確には59.94フレーム。

図2 －4－1 画像更新と表示のタイミングの差のイメージ図

家庭用のTV や、モニタのリフレッシュレートを固定にする場合
1 フレームの時間を固定化する事で作成が楽になる

リフレッシュレートを固定にする事が難しい場合は、
タイマーを用いて、表示周期とは無関係に画像を更新する

2-5 並列動作の概念とは？

並列動作って何？

ゲーム、特にリアルタイム系のゲームでは、複数のキャラクターが画面上を動きまわります。

リアルタイムのゲームでなくとも、アドベンチャーゲームで、メッセージを表示しながらアニメーションをする演出等は良く見かけます。

こういった、複数の処理を同時にこなす事を、並列動作、または並列処理といいます。

もちろん並列処理を行なわなくてもゲームは作れますが、非常に味気なくなる上に、かなりゲームの内容が限られてしまいます。

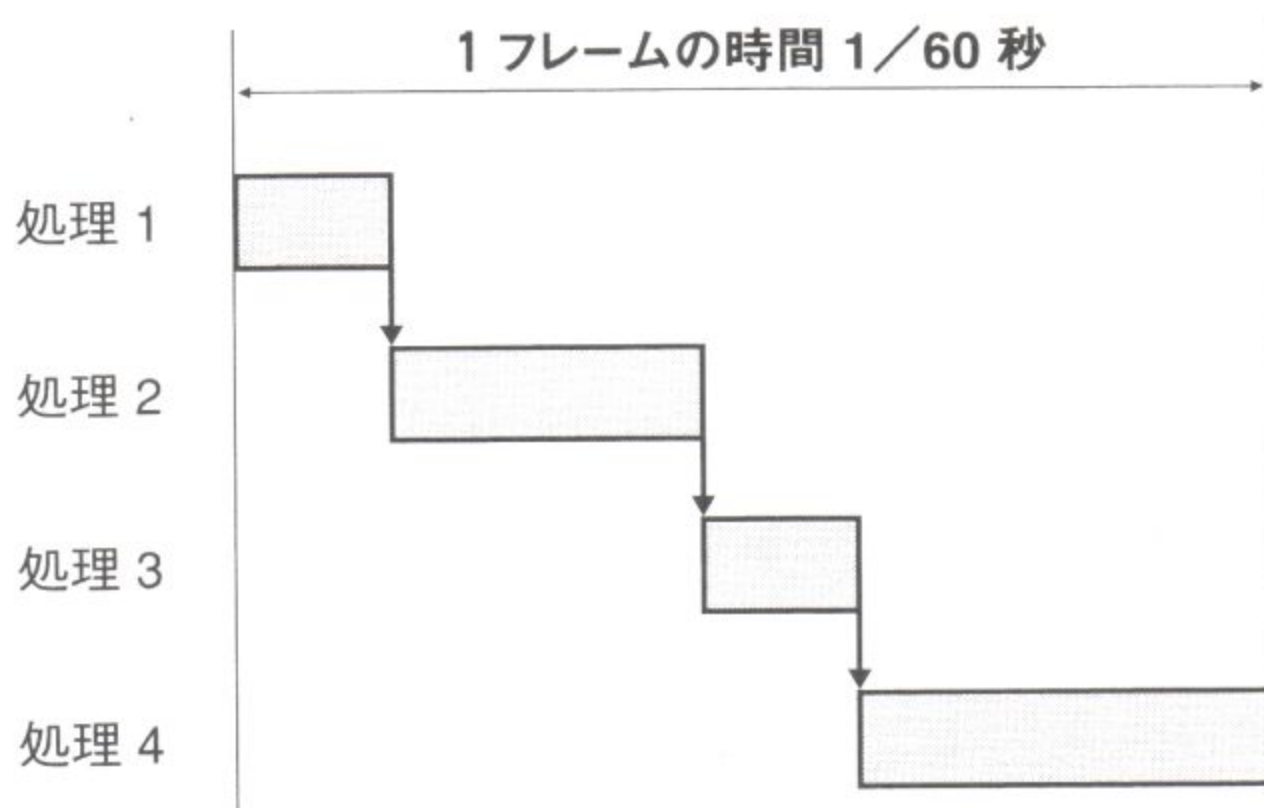
ゲームを作るうえでは事実上、この並列動作が不可欠といってもいいでしょう。

どうやって同時に処理をする？

しかし、並列動作を行なうとはいっても、CPUが1つしかないマシンでは、物理的な並列動作を行なうことはできません。

そこで実際には、1フレームに複数の動作の処理を少しづつ行なう事で、並列動作をしているように見える処理を行ないます。

下図を見てください。

図2－5－1 並列動作の処理の概念図

1フレームの間に少しづつ、処理を行ない表示する事で並列に動いているように見える

このようにすれば、見た目には並列に動作をしているように見えます。

もちろん実際にはそんな事は無く、CPU側から見ると、次々と渡される細かな処理を、逐一実行しているに過ぎません。

2-6 並列動作のシステム

並列動作を管理するためのプログラム

並列動作を行なうためには、それを制御、管理するためにある種のプログラムシステムが必要です。

こういったシステムが無くても並列動作は可能ですが、プログラムのしやすさや開発の効率が格段に変わってきます。

そのため、ゲーム内容に応じて、こういったシステムを搭載する事は事実上、必須になってきているといえるでしょう。

ゲーム用の並列動作システム

この並列動作システムですが、種類は何種類もあり、中にはゲームには向かない物もあります。

もちろんそれとは逆にゲームに特化したシステムも、現場レベルで開発/使用されてきました。
筆者が知りうる範囲ですが、こういったシステムは3系統程存在するようです。

1つは、タスクと呼ばれており、ウェブ上での解説も多いせいか、ゲーム上での並列動作といえばこれを指すことが多いようです。

もう1つは1つはアクト（action または actor の略か？）と呼ばれ、シューティング等で有名なメーカーが作成したと言われています。

最後の1つは呼称は不明ですが、アーケードゲームの基板を設計する際に、CPU メーカーからサンプルとして提供された、OS 用の並列動作システムをゲーム向けに改良したという話が伝わっています。

いずれもメーカーの話は噂レベルですので、真偽の程はわかりませんが、作成した所が全てアーケードゲームメーカーと言うのは、面白い所だと思います。

本書では、その中でも最も有名と思われ、初心者でも扱いやすいタスクシステムを例にとっています。しかし他の2つも基本となる概念は似ており、本書の知識が応用できるでしょう。

2-7 タスクシステムを作成する１

タスクシステムとは

タスクとは、文字通り並列動作を行なうための、個々の小さな処理（タスク）のことです。

この「タスク」は、ゲーム内では、使われる処理の１つの単位として扱われます。

具体的には、１つの弾、１匹の敵、爆発の際の破片等が１つのタスクとなります。

また、上記のような、目に見える処理だけではなく、BGMやスコア管理、ファイルの読み込み等、システム面での処理もこういったタスクを単位として管理する事ができます。

逆に言えば、タスクシステムとは細かな処理でゲームを作りやすくするための管理システムといえるでしょう。

タスクシステムにおけるゲーム作成は、こういった細かな「タスク」の処理を組み合わせる事で行ないます。

タスクシステムの役割

上記のようなシステムのために考えなくてはならないのは、ゲームプログラムを楽に組むためにはどういった処理が望まれているか？　と言う事でしょう。

実は、これはゲームジャンルによってがらっと変わってしまいます。シューティングにはシューティングの、アクションにはアクションに向いた処理システムが存在します。

しかし、ジャンルに特化された部分を除けば基本的な要求は決まっており、それは大体のゲームで共通しています。

1. 並列処理動作の割り当て
2. 個々の処理が使用するワーク（メモリ）の確保と初期化
3. 処理動作の終了とメモリの解放

この部分を共通処理とし、管理システムとしてシステム化できれば、ゲームジャンルに合わせたシステム作成もずいぶんと作成が楽になります。

そしてそれこそが、タスクシステムの目的でもあります。

以下、簡単に解説していきます。

具体的な処理

並列処理動作の割り当て

まず、処理の割り当てです。これは並列動作が目的のシステムである以上当然ですが、いつ実行されるかをユーザー側が管理できなくてはならないため、少々手間が必要です。

メモリの確保と初期化

次に、使用するワークとは、文字通り各処理が使用する作業（ワーク）領域や、記憶しておく情報の事です。

シューティングでは弾の座標や、自機の座標、アクションゲームではキャラのアニメーション情報等がこれに当たります。

処理の終了とメモリの解放

最後に、上記で割り振った処理とメモリの解放です。

これは、大規模なプログラムになるほど重要になってきますが、とりあえずはさほど気にする必要はありません。

ただ、解放のタイミングには注意を要する場合があることだけ覚えておいてください。

※わかりにくい方は、とりあえず処理関数へのポインタと、メモリ（ワーク）をペアで管理する手法、と理解してください。

2-8 タスクシステムを作成する2

処理とメモリの確保

タスクの作成において、まず行なうことは、処理とメモリの確保です。ゲームでは、頻繁にメモリの確保と解放を行なう事が多いため、最初に大きなメモリを確保しておいて、それを自前で管理する事が多いです。

管理には色々な手法がありますが、タスクシステムの場合、通常はリスト構造を使って行なわれます。

リストの内容についてですが、以下の構造体を見てください。

```
//タスク管理領域(今回はタスクそのもの)
struct TCB{
    void            (*Exec)(TCB*);   //タスクを処理する関数
    TCB*            Prev;
    TCB*            Next;
    unsigned int    Flag;
    unsigned int    Prio;
    int             Work[(TASK_WORK_SIZE+3)/4];
};
```

これはタスク管理ブロック(TCB)と呼ばれるものです。この構造体により、メモリと処理の管理を行ないます。

この構造体をみて想像がつくと思いますが、確保できるメモリは固定サイズです。

固定サイズにしている理由は、処理速度の高速化もありますが、可変サイズのメモリ確保によるメモリ管理の複雑さを避けるためです。

固定サイズと聞いて容量等に不安に感じる人もいると思いますが、通常のゲームで大容量のメモリ領域を頻繁に確保/解放するケースは少なく、もし、実際そのようなケースが発生した場合は、ゲームのシステムや状況に合わせて、専用にメモリを確保/解放する事で対処が可能です。

タスクシステムの初期化

さてそれではいよいよ、実際のプログラムを見ていきましょう。
まずはタスクシステムの初期化です。

LIST 2 －8－1

```
//タスクの実体を定義
static  TCB     Task[MAX_TASK_COUNT];

void InitTask()
{
    TCB*      tmpTCB;

    //全タスクを初期化
    ZeroMemory(Task, sizeof(TCB)*MAX_TASK_COUNT);

    tmpTCB = Task;

    //処理関数を格納
    tmpTCB->Exec          = TaskHead;

    //最初のタスクなので最優先で実行
    tmpTCB->Prio          = 0x0000;

    //最初のタスクなので前処理タスクへのリンクは無し
    tmpTCB->Prev          = NULL;

    //初期化時はタスクは１つなので、
    //次に実行するのもこのタスク
    tmpTCB->Next          = Task;

    //タスクが使用中であるフラグ
    tmpTCB->Flag          = _USE;
}
```

最初に静的配列で確保したタスクの領域を初期化します。ここは、単純に０でクリアするだけです。

次に、一番最初に実行されるタスクを作成しています。

タスクを生成するのに必要な情報は タスクを処理する関数へのポインタと、処理実行の優先順位の2つです。

処理関数は、作成者が指定する任意の関数で、優先順位は、最初に実行されるタスクですので0になります。

リストの接続

次はリストの接続ですが、最初のタスクは1つしかないので、接続するタスクには現在作成中のタスクを指定してやります。

後はタスクが使用中であるフラグを設定すれば、初期タスクの作成は終了です。

ここで作成されるタスクはTaskHeadと呼ばれ、処理の先頭を示す重要なタスクです。

また、後のタスクは、全てこのタスク TaskHead から派生して作成されます。

以上でタスクシステムの初期化は終了です。

タスクを実行する処理

最後に実際にタスクが実行される処理を見てみます。

タスクの実行はメインループで行ないます。

OS から終了のメッセージが来ない間はタスク実行関数 TaskExec を常に呼び出し、タスクを実行し続けます。

LIST 2 －8－2

```
//メインループ
while(TRUE){
    //メッセージが来ているか?
    if(PeekMessage(&msg,NULL,0,0,PM_REMOVE)){
    //来ている
        if(msg.message == WM_QUIT) break;    //終了
            TranslateMessage(&msg);
            DispatchMessage(&msg);
    }else{
    //来ていない　この間に主処理
        TaskExec();
    }
}
```

そして実行関数 TaskExec 内では作成されたタスクを処理します。

LIST 2 －8－3

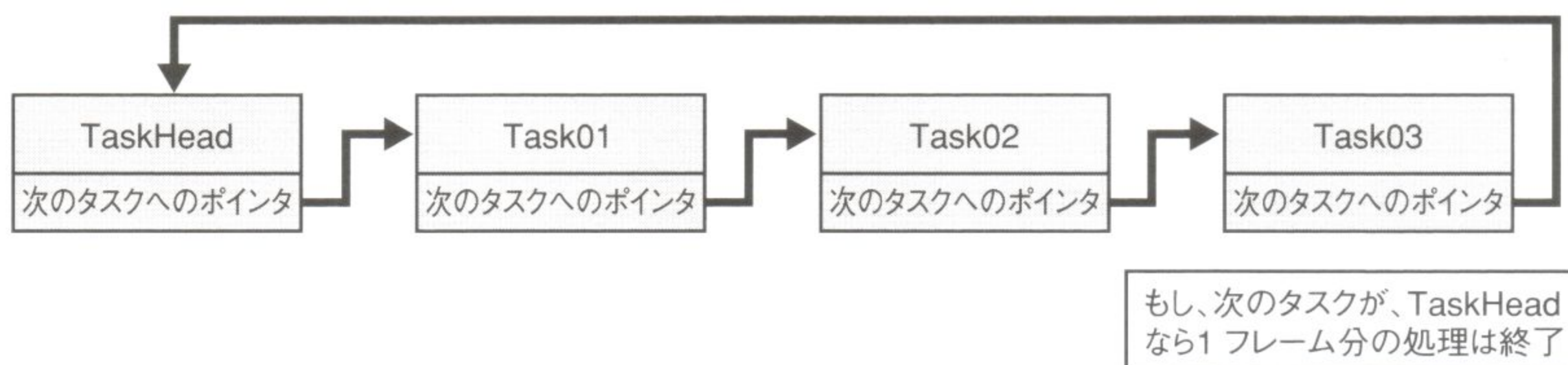
```
void TaskExec( void )
{
    TCB*    execTCB;

    execTCB = Task;
    do{
        execTCB->Exec(execTCB);             //各タスクを処理
        execTCB = execTCB->Next;
    }while(execTCB->Prio != 0x0000);        //リストが一巡したら終了
}
```

各タスクには必ずタスクを処理する関数が登録されていますので、これをLIST構造に登録された順に呼び出し、実行してやります。

リストの終端は次に実行されるタスクが実行優先順位が先頭、すなわちTaskHead かどうかで判断しています。

リストが終端まで達した時、そのフレームで実行されるタスクの処理も終了します。

1フレーム＝1タスクリストの全処理、である事に注意してください。

図2 －8－1 リストをたどってタスクを実行する処理のイメージ図

2-9 タスクシステムを作成する3

タスクの起動と終了

タスクの起動と終了を行なうには、TaskMakeと、TaskKill関数を使用します。

TaskMakeは名前の通りタスクを作成、起動します。

作成のための情報として、タスクを動作させる関数と実行優先順位の2つの引数が必要です。

また、この関数は返り値として、生成されたタスクへのポインタを返します。

逆に起動しているタスクを終了させたい場合は、TaskKillを使用します。

終了させるには、生成されたタスクへのポインタを渡します。

TaskMakeと、TaskKillの処理

では、実際に2つの関数の処理はどうなっているのでしょうか？

まずTaskMakeからみていきましょう。

この関数の内部で行なわれているのは、確保できるタスク（メモリ）の検索と、処理関数の割り当てです。

LIST 2 −9−1

```
TCB* TaskMake(void (*exec)(TCB*),unsigned int prio)
{
    TCB*    newTCB;
    TCB*    prevTCB;
    TCB*    nextTCB;
    int     id;

    //空いているタスク領域を探す
    for(id=0; id<MAX_TASK_COUNT; id++){
        if(!(Task[id].Flag & _USE))break;
    }
    //空き領域が無い…　生成失敗
    if(id == MAX_TASK_COUNT)return NULL;

    newTCB = &Task[id]; //生成成功！

```

```
//挿入する優先順位を検索
    prevTCB = Task;
    //タスクの終端までを検索
    while(prevTCB->Next->Prio != 0x0000){
        if(prevTCB->Next->Prio > prio)break;
        prevTCB = prevTCB->Next;
    }
    nextTCB = prevTCB->Next;


    ZeroMemory(newTCB, sizeof(TCB));                //0で初期化

    newTCB->Exec            = exec;
    newTCB->Prio            = prio;

    //タスクリストへの挿入
    newTCB->Prev            = prevTCB;
    newTCB->Next            = nextTCB;
    prevTCB->Next           = newTCB;
    nextTCB->Prev           = newTCB;

    newTCB->Flag            = _USE;                 //生成完了

    return newTCB;
}
```

まず、空いているタスク(メモリ)を探します。タスクが空きかどうかを知るには使用中かどうかのフラグで確認します。

もし空きタスクがない場合は、エラーを返して関数は終了します。

タスクが見つかったら、次にリストにタスクを登録します。

リストをたどっていき、処理優先順位にあわせてタスクの挿入位置を探します。

このときもし、登録するタスクの挿入位置が0ならば、タスクの最終位置なので、その位置を挿入位置とします。

図2－9－1 挿入位置のイメージ図

前のタスクへのポインタ
TaskHead
優先順位0
次のタスクへのポインタ

前のタスクへのポインタ
Task01
優先順位1000
次のタスクへのポインタ

前のタスクへのポインタ
Task02
優先順位2000
次のタスクへのポインタ

優先順位を検索して、
新たなタスクの挿入位置を探す

前のタスクへのポインタ
新規作成のTask
優先順位1500
次のタスクへのポインタ

新規タスクの挿入位置

前のタスクへのポインタ
TaskHead
優先順位0
次のタスクへのポインタ

前のタスクへのポインタ
Task01
優先順位1000
次のタスクへのポインタ

前のタスクへのポインタ
新規作成のTask
優先順位1500
次のタスクへのポインタ

前のタスクへのポインタ
Task02
優先順位2000
次のタスクへのポインタ

前のタスクへのポインタ
TaskHead
優先順位0
次のタスクへのポインタ

前のタスクへのポインタ
TaskHead
優先順位0
次のタスクへのポインタ

前のタスクへのポインタ
新規作成のTask
次のタスクへのポインタ

タスクリストの初期状態
もし次タスクがタスクヘッドなら最終位置に挿入

挿入位置が決まったら、タスクの実行関数へのポインタと処理優先順位を登録し、挿入位置にタスクを登録します。

その後、タスクが使用中であることを示すフラグを設定して処理は終了です。

最後に、返り値として生成したタスクへのポインタを返します。

この関数を呼び出したタスクはこのポインタを利用して、新たなタスクにデータを受け渡したり、処理を行なわせます。

実際には管理のために、メモリとタスクの確保は分けられる事も多いのですが、このシステムでは同一の物として扱っています。

次にTaskKillですが、こちらは特に複雑な事はしていません。
指定されたタスクをリストからはずし、タスク(メモリ)を解放します。
あとは使用フラグを消去するだけで処理は終了です。

LIST 2 −9−2

```
void TaskKill(TCB* killTCB)
{
    killTCB->Prev->Next = killTCB->Next;
    killTCB->Next->Prev = killTCB->Prev;
    //消去
    killTCB->Flag       = 0;
}
```

2-10 タスクシステムの使い方

タスクシステムを使ってみる

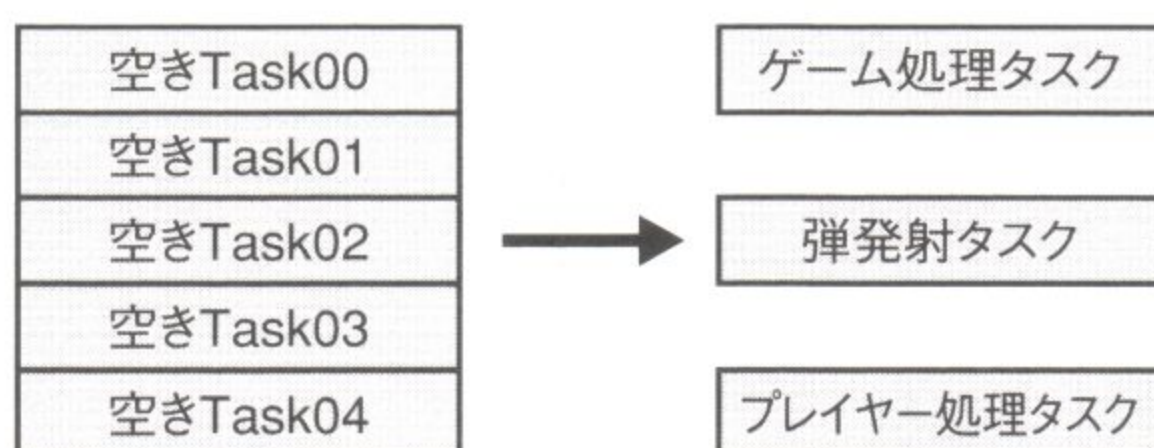
タスクシステムを使用してみましょう。

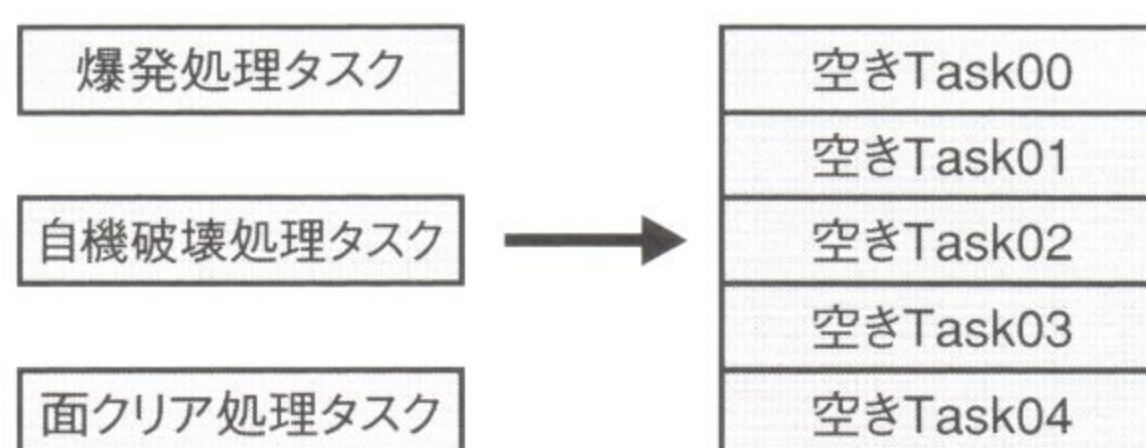
[2―7]でも触れていますが、タスクシステムは処理とメモリをペアで管理します。

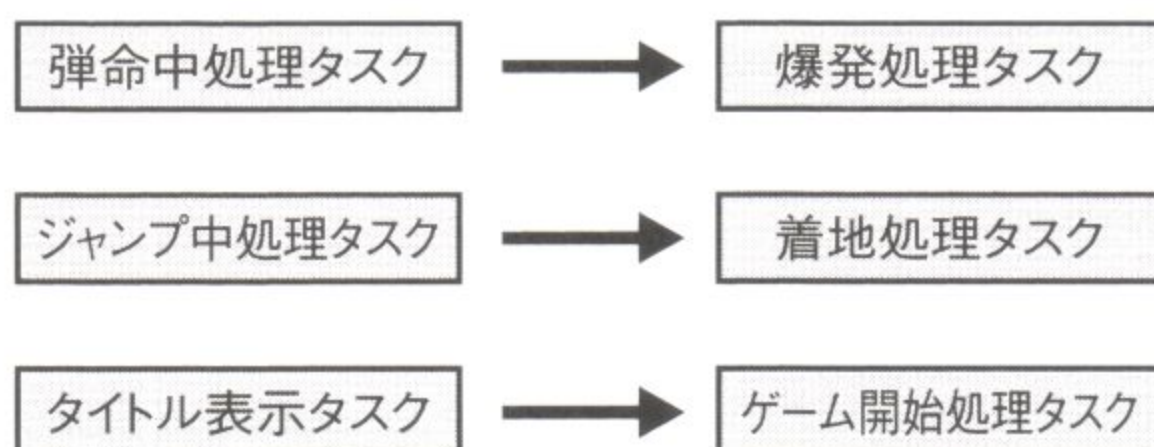
この管理は大きく分けて「確保」「解放」「処理切り替え」「通信」の4つからできています。

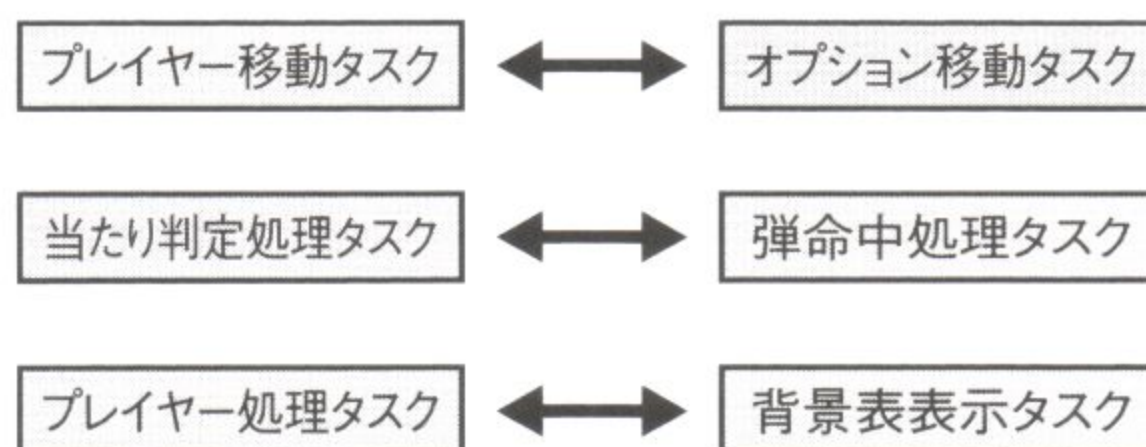
図2―10―1 「確保」「解放」「処理切り替え」「通信」のイメージ図

確保
使用されていないタスクを指定された処理に割り当てる

空きTask00		ゲーム処理タスク
空きTask01		
空きTask02	→	弾発射タスク
空きTask03		
空きTask04		プレイヤー処理タスク

解放
確保とは逆に処理が終了したタスクを空きタスクに戻す

爆発処理タスク		空きTask00
		空きTask01
自機破壊処理タスク	→	空きTask02
		空きTask03
面クリア処理タスク		空きTask04

処理切り替え
特定の処理後に別の処理に切り替える

弾命中処理タスク	→	爆発処理タスク
ジャンプ中処理タスク	→	着地処理タスク
タイトル表示タスク	→	ゲーム開始処理タスク

通信
タスク同士で相互に情報のやり取りを行う

プレイヤー移動タスク	↔	オプション移動タスク
当たり判定処理タスク	↔	弾命中処理タスク
プレイヤー処理タスク	↔	背景表示タスク

確保

まず、確保ですが、これはTaskMakeを使います。

```
TaskMake(GameMain,0x1000);          //ゲーム管理用のタスクを作成
```

最初の引数は、タスクの処理をする関数、2つ目は実行の優先順位です。

この関数を呼ぶだけで、タスクが作成、起動されます。

この時、この関数を呼んだタスクを「親タスク」、作成されたタスクを「子タスク」と呼びます。

また、この時作成される関数の型は固定です。

```
void GameMain(TCB* thisTCB);
```

引数として、そのタスクへのTCBが渡されます。

TCBが違えば、当然違うタスクですので、特定のTCBに頼った処理の記述は避けなくてはいけません。

解放

次は解放です。これはTaskKillを使用します。引数には解放するTCBへのポインタを指定します。

```
//生成後、処理終了
TaskKill( thisTCB );
```

この関数を呼ぶ時には、他のタスクと情報のやり取り(通信)や管理をしていないか十分に注意して解放してください。

管理中に解放すると、非常に厄介なバグのもとになります。

処理切り替え

次に処理切り替えは、TaskChange を使用します。

通常タスクは、TaskMake時に指定された関数で呼ばれ続けますが、この関数で、呼び出される関数を切り替える事ができます。

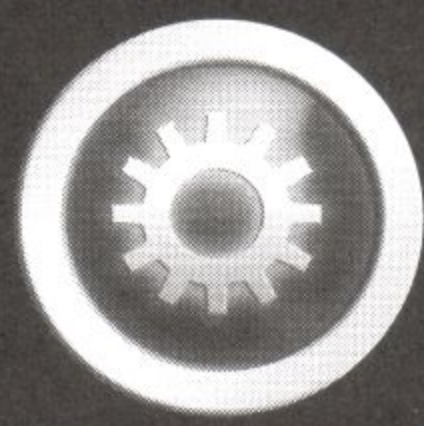
```
//処理の切り替え、但し実際処理が切り替わるのは1フレーム後
TaskChange(thisTCB,TaskHead00);
```

指定には、切り替える処理関数へのポインタと、TCBへのポインタを指定します。

TCBを指定するのは、管理するタスクが外部から処理を切り替えられるようにするためです。

通信

最後に通信ですが、実はこの部分は実装していません。
というのも、複雑なゲームで無い限り、本格的な通信機能はあまり必要とされないためです。
通常タスク間の通信は、各タスクへのポインタを使用して直接やり取りします。

```
//作成時に子タスクへのポインタを保持しておいて
tmp_tcb = TaskMake( GameProc, 0x2000 );
//直接 やり取りする
tmp_tcb->Work[0] = 255;
//子タスクから親タスクと通信するために、親タスクへのポインタを渡しておく
(TCB*)tmp_tcb->Work[0] = thisTCB;
```

Chapter 3

システム

3-1 複数のキャラクターを動かす

ここでは、第２章で解説した並列動作処理を利用して、複数のキャラクターを動かす手順を示します。

システムを使用する上で基本的な事ですが、手順や概念を理解するまでは、分かりにくい処理でもありますので、１つ１つ理解してください。

並列処理システムの初期化

まずは初期化を行ないます。初期化関数init03_01で使用するビットマップを読み込んでいます※。

この初期化関数自体は、プログラムが開始される時に、システム側で作成されたタスクGameMainから１度だけ呼び出されます。

初期化終了後、GameMainは、メイン処理の実行タスクとして処理を切り替えられ、通常のゲーム処理に処理を移行します。

この初期化からゲームへの処理の流れは、初期化が不要な時などの例外を除いて、本書のプログラムでは共通の物です。

以下にプログラムを示します。

LIST 3－1－1 GameMainのプログラム

```
void GameMain(TCB* thisTCB)
{
static int select_chapter = 0;
    char str[128];
    RECT font_pos = {0,0, 640, 480, };

    //メニュー選択
    if( g_DownInputBuff & KEY_RIGHT )
    {
        select_chapter++;
        if(select_chapter >= FUNC_COUNT) select_chapter = 0;
    }
    if( g_DownInputBuff & KEY_LEFT )
```

※詳細は第５章を参照。

```
    {
        select_chapter--;
        if(select_chapter < 0) select_chapter = FUNC_COUNT-1;
    }

    if( g_DownInputBuff & KEY_Z      )
    {//Zボタンで選択した処理の実行
        if(chapter_table[select_chapter].InitFunc != NULL)
            //開始時に1度だけ実行する関数
            chapter_table[select_chapter].InitFunc(thisTCB);
        //毎フレーム実行する関数
        TaskChange(thisTCB,chapter_table[select_chapter].ExecFunc);
    }

    //サンプル名の表示
    font_pos.top   = SCREEN_HEIGHT/2;
    sprintf( str,"%s",chapter_table[ select_chapter ].ChapterName);
    g_pFont->DrawText( NULL, str, -1, &font_pos, DT_CENTER, 0xffffffff);

    font_pos.top   = SCREEN_HEIGHT/3 * 2;
    g_pFont->DrawText( NULL, "方向キーでメニュー選択¥n Zキーで実行", -1, &font_pos,
DT_CENTER, 0xffffffff);
}
```

メイン処理

次にメインの処理です。これはタスク用の関数exec03_01で行なわれています。

処理内容によっては、この関数だけですむ場合もありますが、並列動作を行なう場合は、大抵複数の関数を作成します。

サンプルプログラムでは、並列動作の作成例として2種類のタスクを生成しています。

では、プログラムを追って行きましょう。

タスクを作成

まず最初に、タスク生成関数TaskMakeを呼び出して単一のタスクexec03_01_raderを作成しています。

この関数はタスクワークへのポインタを返しますので、作成したタスクを外部から利用する場合は、このポインタを記録しておきます。

次に、ループを使用して同種類のタスク exec03_01_move を一度に複数作成しています。

ここでも同様に TaskMake を使用してタスクを作成し、ポインタを記録しています。

そして作成したタスクを外部から利用するために、記録したポインタを始めに作成したタスク exec03_01_rader に登録しています。

その後、このタスクに必要な初期化処理を行なっていますが、ここでは作成例の紹介だけに留め、処理内容については後述※します。

自己消滅

最後に、このタスクは生成処理専用なので、自己消滅をして終了します。

もし、このタスクで他のタスクを管理したいのであれば、TaskChange を用いて処理を切り替えるとよいでしょう。

LIST 3 －1－2

```
void init03_01(TCB* thisTCB)
{
    //使用するテクスチャの読み込み
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥0055.png",&g_pTex[0] );
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥PIX.png",&g_pTex[1] );
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥RADER.png",&g_pTex[2] );
}

void exec03_01(TCB* thisTCB)
{

#define ADD_X 8.0
#define ADD_Y 8.0
#define ADD_SPEED 20.0
    TCB* pnew_task;
    TCB* prader_task;
    EX3_1_STRUCT* pnew_work;
    EX3_1_STRUCT* pEX3_1work = (EX3_1_STRUCT*)thisTCB->Work;
    EX3_1_STRUCT* pEX3_rader;

    int loop;
    //レーダー処理の作成
    prader_task = TaskMake(exec03_01_rader, 0x8000);
```

※[3－9][3－14]で解説しています。

```
    //複数表示処理の初期化と起動
    for(loop = 0;loop < EX3_1_MOVE_COUNT; loop++)
    {
        pnew_task = TaskMake(exec03_01_move, 0x2000);
        pnew_work = (EX3_1_STRUCT*)pnew_task->Work;

        //レーダー処理の為に登録
        prader_task->Work[ loop ] = (int)pnew_work;


        pnew_work->AddX  = (((rand() % 100) / 100.0) - 0.5) * ADD_SPEED;
        pnew_work->AddY  = (((rand() % 100) / 100.0) - 0.5) * ADD_SPEED;

        pnew_work->sprt.X = SCREEN_WIDTH / 2 - 32;              //初期座標
        pnew_work->sprt.Y = SCREEN_HEIGHT / 2 - 32;
    }

    TaskKill(thisTCB);
}
```

3-2 文字の表示

フォントオブジェクトを作成する

次は文字の表示です。

DirectX には文字表示を行なうための、フォントオブジェクトを作成する API、D3DXCreateFont が用意されています。

フォントの作成方法

フォントの作成方法は簡単で、必要なものは表示する IDirect3DDevice9 へのポインタと、作成されるフォントを格納するポインタです。

ただ、この API は引数が多く、細かく指定しようとすると結構面倒です。

ここでは文字の大きさ以外は可能な限りデフォルトの引数を使っています。

```
D3DXCreateFont(g_pD3Ddevice,
 16,                                  //幅
NULL,                                 //高さ
FW_DONTCARE,                          //太さ
NULL,                                 //ミップマップレベル
FALSE,                                //斜体
SHIFTJIS_CHARSET,                     //文字セット
OUT_DEFAULT_PRECIS,                   //出力精度
DEFAULT_QUALITY,                      //出力品質
DEFAULT_PITCH | FF_DONTCARE,          //フォントピッチ＆フォントファミリ
"MS ゴシック",                        //フォント名
&g_pFont);
```

最初と最後の引数が、それぞれ DirectX のデバイスとフォント格納のポインタです。

あとは、このフォントの DrawText メソッドを 使用して表示を行ないます。

こちらも表示をするための引数が多いですが、ほとんどは固定の引数で問題ありません。

```
DrawText( NULL,                       //高速化のための D3DXSprite 構造体へのポインタ
                                      （NULL で使用しない）
"Hello DirectX!",                     //表示する文字列
```

-1,	//表示する文字数(-1で最後まで表示)
pRect,	//表示する座標領域のRECT構造体へのポインタ
DT_LEFT,	//文字位置(左詰)
0xffffffff	//表示色
);	

使う際の注意点

注意点として、このメソッドを使用する時は必ずIDirect3DDevice9のメソッドBeginSceneと、EndSceneの間で使用しなくてはなりません。

また、この文字表示はフォントは綺麗なのですが、処理が非常に重く、処理負荷の大きいゲームにはあまり向きません。

ノベル系のゲームやデバッグ時の表示に使うと良いでしょう。

3-3 ステージやシーンを切り替える1

「シーン」とは

通常、ゲームは様々なシーンに分ける事が出来ます。

タイトルから始まり、ゲーム中、オプション画面、ゲームオーバー、エンディング等々。

これはプレイヤーに、シーンの切り替えを伝える意味もありますが、プログラムでの管理を行なう上でも重要な意味を持ちます。

ここでは、シーンの切り替えとその管理を考えてみましょう。

図3－3－1 ゲームは様々なシーンに分けられるイメージ図

プログラマの視点から考える「シーン」

さて、シーンとはプログラム上において一体どういうものを言うのでしょうか？

これは、ゲームによって定義は変わってしまうのですが、管理の面から考えると、「入力とそれに対する処理をグループ化したもの」とするのが良いかと思います。

もう少し砕けた言い方をすると、入力方法とその処理が根本的に変わってしまう場所の事です。

例えば、ゲーム中ではキー入力によって移動したり、弾を撃ったりしますが、タイトル画面やデモシーンでは同じ操作をしても、まったく別の動作になります。

これらが変わってしまうポイントをシーンの切り替えと定義します。

では、実際の管理ですが、これは並列動作の処理をうまく使う事で実現できます。

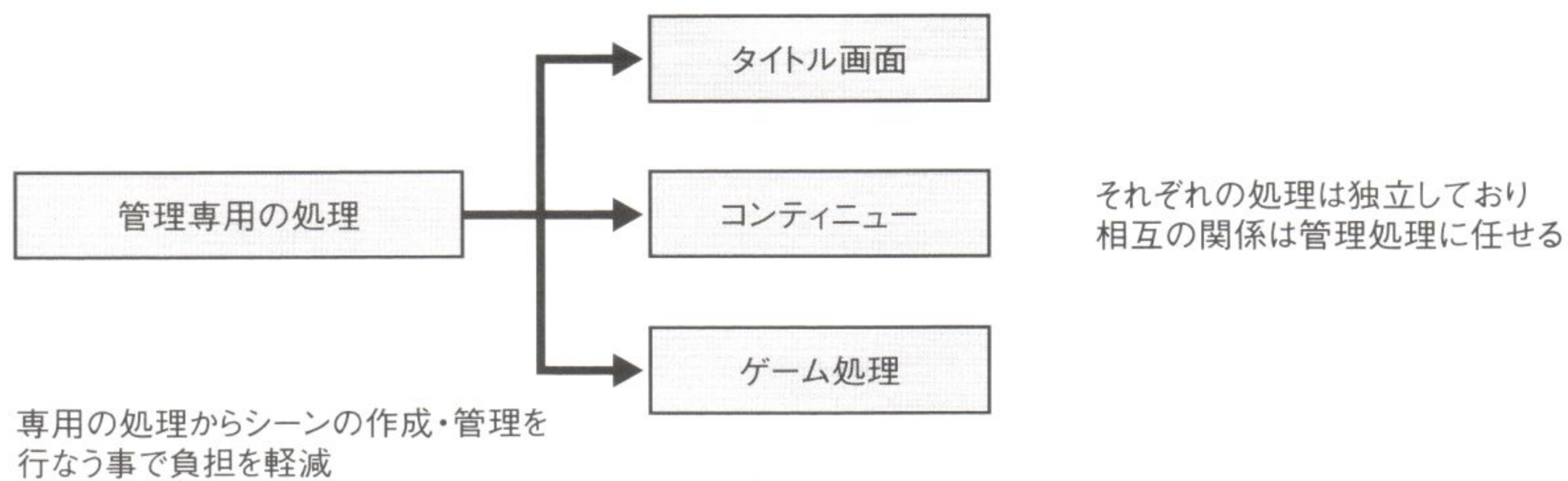
すなわちシーン管理専用の処理を作成して、面倒な処理はそこに任せてしまうのです。

こうする事で、処理の終了時以外は、ほとんど切り替えを意識しなくてすみます。

図3 －3－2 並列処理で、ゲームのシーンを管理するイメージ図

管理専用の処理
タイトル画面
コンティニュー
ゲーム処理

専用の処理からシーンの作成・管理を
行なう事で負担を軽減

それぞれの処理は独立しており
相互の関係は管理処理に任せる

3-4 ステージやシーンを切り替える2

シーン管理の手法

では、実際の管理処理を見ていきましょう。

まずは管理用のタスクの初期化ですが、サンプルでは、初期化処理自体もシーンの一部に含めています。

そのため、ここではシーン管理用の変数に初期化のシーンIDを代入するだけです。

なお、シーンとその切り替えは管理IDで管理を行なっており、値はマクロで定義されています。

```
#define SCENE_INIT          -1      //初期化
#define SCENE_CHANGE_WAIT   0       //切り替え時の待ち状態
#define SCENE_TITLE         1       //タイトルシーン
#define SCENE_GAME          2       //ゲーム中
#define SCENE_GAME_OVER     3       //ゲームオーバー
#define SCENE_OPTION        4       //オプション画面
#define SCENE_END           5       //シーンが終了した時
```

シーン切り替えは他の処理からの通信で

さて、メインの管理処理ですが、変数SceneIDによって切り替えの処理を行ないます。

処理はシーンに応じてswitch文で行ない、毎フレーム、シーンのチェックを行ないます。

ですが、この変数は内部的なもので、通常、処理の切り替えを指示するのは外部の処理です。

そのため、何らかの方法を用いて、他の処理との通信を行なう必要があります。

ここでは、様々な処理からアクセスがあることを考え、グローバル変数で通信を行なっています。

通常は悪役のように言われるグローバル変数ですが、目的をもって管理の下に使用するのであれば、大きな問題は起こらず、非常に便利に使用する事が出来ます。

ただ、グローバル変数に抵抗を感じるのであれば、何らかの通信システムを用意して代替しても良いでしょう。

グローバル変数を使った通信処理

その通信処理ですが、この管理タスクで、常にその内容を見張り続ける事で行ないます。

通常、通信用のグローバル変数g_SCENE_CHANGE_COMMには切り替え待ちを示す値として SCENE_CHANGE_WAIT が代入されています。

もしこの値が他の値になれば、切り替えが外部の処理から指示されたと言う事です。

この時の値が切り替え処理を行なうシーンの管理IDになります。

なぜこのようにしているかというと、シーン内部の処理内容によって、次に処理を行なうシーンが変わる事があるためです。

例えば、コンティニューの処理で、シーン中に継続を選択したかどうかで、切り替わるシーンが変わります。

そのため、そのシーンが終了した時に、どのシーンへ切り替わるのかの指示をシーンの処理先で行なう必要があるのです。

シーンの終了

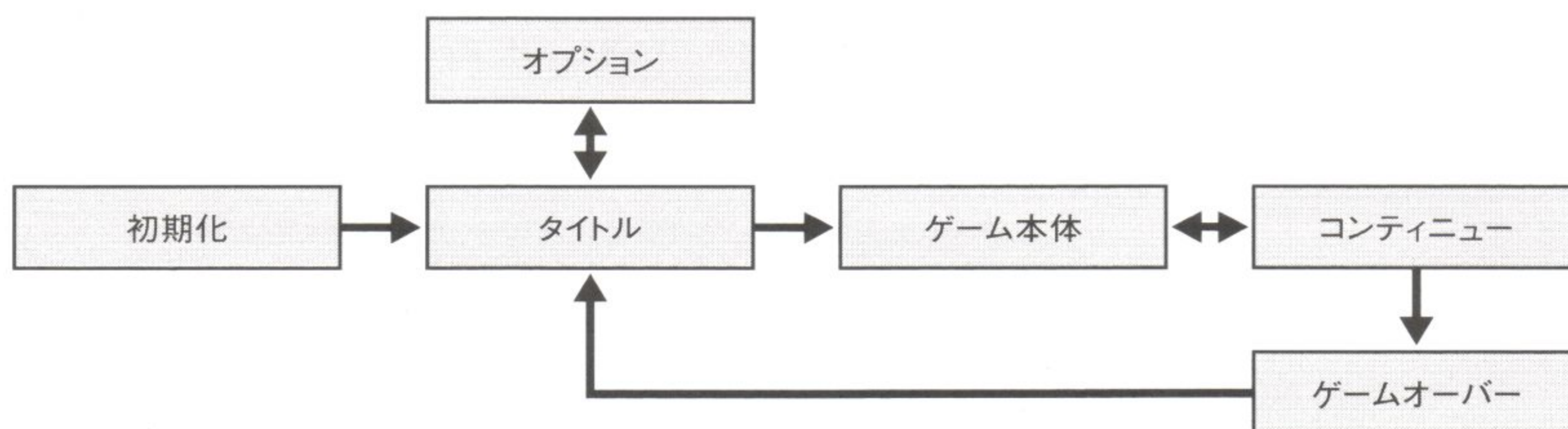
さて、実際の切り替え処理ですが、リクエストされたシーンに合わせて、そのシーンの処理を作成します。

この時、本来はリクエストされた元のシーン(例えばゲーム中からゲームオーバーになった時の、元のゲーム中のシーン)を終了させる必要があります。

ですが、終了処理は手続きが面倒だったり時間が掛かるケースがあるため、ここでは元のシーンが自前で終了処理を行なうようにしています。

作成後、内部のシーン変数を指定のシーンに切り替えて、処理は再度切り替え待ち状態に入ります。

管理側の処理は以上となります。少々ややこしいですが、ここさえ作成しておけば、シーン処理側の作成はずっと楽になります。

図3−4−1 シーン切り替え相関図

LIST 3 －4－1 exec03_SCENE_CTRL

```
void init03_SCENE_CTRL(TCB* thisTCB)
{
    EX3_SCENE_STRUCT* pEX3_SCENEwork;
    pEX3_SCENEwork = (EX3_SCENE_STRUCT*)thisTCB->Work;

    //初期化中の状態
     pEX3_SCENEwork->SceneID = SCENE_INIT;
}
void exec03_SCENE_CTRL(TCB* thisTCB)
{
    EX3_SCENE_STRUCT* pEX3_SCENEwork;
    pEX3_SCENEwork = (EX3_SCENE_STRUCT*)thisTCB->Work;

    switch( pEX3_SCENEwork->SceneID )
    {
        case SCENE_INIT: //初期化中
                pEX3_SCENEwork->SceneID = SCENE_TITLE;
                TaskMake( exec03_SCENE_TITLE, 0x2000 );
                g_SCENE_CHANGE_COMM = SCENE_CHANGE_WAIT;
                break;
        case SCENE_TITLE: //タイトル画面
                if( g_SCENE_CHANGE_COMM != SCENE_CHANGE_WAIT )
                {
                    if( g_SCENE_CHANGE_COMM == SCENE_GAME )
                    {
                        TaskMake( exec03_SCENE_GAME, 0x2000 );
                        pEX3_SCENEwork->SceneID = SCENE_GAME;
                    } else
                    if( g_SCENE_CHANGE_COMM == SCENE_OPTION )
                    {
                        TaskMake( exec03_SCENE_OPTION, 0x2000 );
                        pEX3_SCENEwork->SceneID = SCENE_OPTION;
                    }
                        g_SCENE_CHANGE_COMM = SCENE_CHANGE_WAIT;
                }
                break;
        case SCENE_GAME: //ゲーム処理
                if( g_SCENE_CHANGE_COMM != SCENE_CHANGE_WAIT )
                {
```

```
                pEX3_SCENEwork->SceneID = SCENE_GAME_OVER;
                TaskMake( exec03_SCENE_GAME_OVER, 0x2000 );
                g_SCENE_CHANGE_COMM = SCENE_CHANGE_WAIT;
            }
            break;
        case SCENE_GAME_OVER:  //ゲームオーバー処理
            if( g_SCENE_CHANGE_COMM != SCENE_CHANGE_WAIT )
            {
                if( g_SCENE_CHANGE_COMM == SCENE_GAME )
                {
                    pEX3_SCENEwork->SceneID = SCENE_GAME;
                    TaskMake( exec03_SCENE_GAME, 0x2000 );
                } else
                if( g_SCENE_CHANGE_COMM == SCENE_TITLE )
                {
                    pEX3_SCENEwork->SceneID = SCENE_TITLE;
                    TaskMake( exec03_SCENE_TITLE, 0x2000 );
                }
                g_SCENE_CHANGE_COMM = SCENE_CHANGE_WAIT;
            }
            break;
        case SCENE_OPTION:  //オプション画面
            if( g_SCENE_CHANGE_COMM != SCENE_CHANGE_WAIT )
            {
                pEX3_SCENEwork->SceneID = SCENE_TITLE;
                TaskMake( exec03_SCENE_TITLE, 0x2000 );
                g_SCENE_CHANGE_COMM = SCENE_CHANGE_WAIT;
            }
            break;
    }
}
```

3-5 タイトル画面

タイトル画面での処理

タイトル画面はゲームの顔とも言える部分です。

ゲームの開始処理はもちろんここから始まりますが、シーン終了後の切り替えもタイトル画面になる事が多くなります。

オープニング終了後、ゲームオーバーの後、エンディングの後等、処理の終了後にタイトルに戻るシーンはかなりあります。

そのため、このシーンは見た目と同等か、それ以上に様々なシーンの中継地点になります。

また、隠しコマンドで追加ステージが現れたり、ロード画面への処理も行なったりなど、入力が集中するシーンでもあるため、意外と処理が集中するシーンでもあります。

もっとも、よほど凝ろうとさえしなければ、画像の表示と開始処理を行なうだけですので、処理は非常に単純になります。

タイトル画面のプログラミング

では切り替え処理の解説も兼ねて、実際のプログラムを見ていきましょう。

サンプルでは、ゲームの開始処理とオプション画面への切り替え、そしてタイトル文字の表示を行なっています。

切り替え処理処理自体は、キー入力に合わせて、次に実行されるシーンのIDをシーン切り替えを通達するグローバル変数に代入し、自己の処理を終了してやるだけです。

シーン管理自体は、[3－3]で解説した様にシステム側で行なっていますので、非常にシンプルな処理になります。

ここでの切り替え先はゲームの主処理とオプション画面だけですが、条件や隠しコマンド等で、切り替え先を追加するのはさほど難しく無いでしょう。

LIST 3－5－1 exec03_SCENE_TITLE

```
void exec03_SCENE_TITLE(TCB* thisTCB)
{
    EX3_SCENE_STRUCT* pEX3_SCENEwork;
    pEX3_SCENEwork = (EX3_SCENE_STRUCT*)thisTCB->Work;

```

```
        FontPrint( 176, 160,"GAME TITLE SCREEN");

        //一定時間ごとに点滅を繰り返す
        pEX3_SCENEwork->Time++;
        if( pEX3_SCENEwork->Time & 0x10)
        {
          FontPrint( 176, 256,"PUSH 'Z' GAME START");
          FontPrint( 176, 272,"PUSH 'X' OPTION");
        }

        //シーンの切り替えをシーン管理タスクに伝える
        if( g_DownInputBuff & KEY_Z )
        {
                //ゲームを選択
                g_SCENE_CHANGE_COMM = SCENE_GAME;
                TaskKill(thisTCB);
                return;
        }
        if( g_DownInputBuff & KEY_X )
        {
                //オプションを選択
                g_SCENE_CHANGE_COMM = SCENE_OPTION;
                TaskKill(thisTCB);
                return;
        }
}
```

3-6 ゲームオーバーを作成する

なぜゲームオーバー画面がある？

ゲームオーバーの処理を考えてみます。

通常ゲームが終了すると、即座にタイトル画面には行かず、ゲームオーバーの画面を表示します。

これは幾つか理由がありますが、最大の理由はプレイヤーにゲームオーバーになった事を「確実に納得」してもらうためです。

自機が爆発した直後にいきなりタイトル画面に戻ったのでは、プレイヤーは何が起こったのか一瞬理解できず、ゲームへの不満へとつながってしまいます。

そのため、市販のゲームではゲームオーバー画面と同時に、ゲームオーバーの理由や攻略のヒントを表示するゲームなども見かけます。

また別の理由としては、コンティニューを行なうゲームの場合、ここでコンティニュー選択の処理が可能になる事も挙げられます。

ゲームオーバーの処理

では、実際のサンプルを見ていきましょう。

まず、ゲームオーバーの表示処理を行ない、その後コンティニューの選択処理に入ります。

選択はZ、Xの直接キー入力で行ない、入力に応じてゲーム処理、またはタイトル画面へシーンを切り替えるようにします。

この部分の実際の処理は、シーン管理処理に任せられるため、ずっとシンプルになっています。

もしコンティニュー処理が不要な場合は、一定時間ゲームオーバーを表示する処理などに差し替えると良いでしょう。

LIST 3－6－1 exec03_GAME_OVER

```
void exec03_SCENE_GAME_OVER(TCB* thisTCB)
{
    EX3_SCENE_STRUCT* pEX3_SCENEwork;
    pEX3_SCENEwork = (EX3_SCENE_STRUCT*)thisTCB->Work;

     FontPrint( 208, 160,"GAME OVER");
```

```
    //一定時間ごとに点滅を繰り返す
    pEX3_SCENEwork->Time++;
    if( pEX3_SCENEwork->Time & 0x10)
    {
      FontPrint( 176, 256,"PUSH 'Z' CONTINUE");
      FontPrint( 176, 272,"PUSH 'X' TITLE");
    }

    //シーンの切り替えをシーン管理タスクに伝える
    if( g_DownInputBuff & KEY_Z )
    {
        //ゲームを続行する
        g_SCENE_CHANGE_COMM = SCENE_GAME;
        TaskKill(thisTCB);
        return;
    }
    if( g_DownInputBuff & KEY_X )
    {
        //タイトルへ戻る
        g_SCENE_CHANGE_COMM = SCENE_TITLE;
        TaskKill(thisTCB);
        return;
    }
}
```

3-7　オプション画面を作る

オプション画面での処理

オプション画面とは、言うまでもなくゲームの設定を行なう画面の事です。

ここで言う設定とは、ゲームの難易度やプレイヤーの数を変更する事を指します。

また、ゲームによっては、使用する言語の設定やキー操作の設定などもここで行なわれます。

このように、色々な設定を行なう画面ではありますが、実は、処理自体はさほど難しい事はありません。

それは基本的に、この画面で行なうのは、あくまで数値やフラグの設定が主で、実際の処理は別の部分で行なう事が大半だからです。

オプション画面のプログラミング

サンプルリストを見てください。これはオプション画面でランク設定を行なうプログラムです。

表示部分を除けば、変数RANKの増減を管理しているだけです。

実際の処理は、RANKを使用する他のプログラムが行ないます。

ここでは、RANKはstatic変数を用いていますが、他の処理から利用できるように、実際にはグローバル変数か、関数呼び出しによる設定処理プログラムを使用する事になるでしょう※。

他処理からの呼び出し

サンプルではオプション画面は、タイトル画面から呼び出される様になっています。

しかし実際のゲームでは、タイトル以外、特にゲーム中でもオプション設定ができるゲームがあります。

ユーザーの視点から見てもこちらの方が親切といえますので、可能であれば、何処から呼び出しても設定が可能な画面にすると良いでしょう。

LIST 3－7－1

```
void exec03_SCENE_OPTION(TCB* thisTCB)
{
#define RANK_MAX 5
    EX3_SCENE_STRUCT* pEX3_SCENEwork;
    pEX3_SCENEwork = (EX3_SCENE_STRUCT*)thisTCB->Work;
```

※設定処理という性質からすると、グローバル変数はなるべく避け、設定関数を作成した方が良いでしょう。

```
    char str[128];

//ゲーム中のランク
//本来は外部から参照できるようグローバル変数にするか
//関数による処理にする
static int RANK = 2;


    //ランクの増減処理
    if( g_DownInputBuff & KEY_RIGHT ) RANK++;
    if( g_DownInputBuff & KEY_LEFT  ) RANK--;
    //ランク上限値、下限値チェック
    if( RANK >= RANK_MAX ) RANK = RANK_MAX-1;
    if( RANK <  0        ) RANK = 0;



     //表示処理
     FontPrint( 208, 160,"GAME OPTION");
     //ランクの表示
     sprintf( str, "SELECT RANK %d",RANK+1);
     FontPrint( 208, 208,str);

    //一定時間ごとに点滅を繰り返す
    pEX3_SCENEwork->Time++;
    if( pEX3_SCENEwork->Time & 0x10)
    {
      FontPrint( 176, 240,"PUSH 'X' OPTION EXIT");
    }

    //シーンの切り替えをシーン管理タスクに伝える
    if( g_DownInputBuff & KEY_X )
    {
        g_SCENE_CHANGE_COMM = SCENE_END;
        TaskKill(thisTCB);
        return;
    }
}
```

3-8 自分でフォントを作る

自作フォントの利点

自分で作成したフォントをゲームで使用してみましょう。

文字を表示する場合、Windowsに搭載されているフォントでも良いのですが、作成したゲーム専用のフォントを使用するとずっと雰囲気が良くなります。

また、WindowsAPIのフォント表示に比べてとても高速なので、処理速度が気になる場合にも有効です。

表示方法

さて表示方法ですが、まず、フォントのグラフィックデータを用意します。

そして、スプライトのグラフィックデータとして扱い、フォントをキャラクターとして表示してやるのです。

グラフィックデータですので差し替えや切り替えも容易ですし、何よりスプライトですのでとても高速です。

フォントデータの管理方法

なお、用意するグラフィックデータですが、文字1つで1枚のグラフィックデータとすると分かりやすいのですが、データの管理が大変になってしまいます。

そこでここでは、1枚のグラフィックに複数のフォントを格納する方法を紹介します。

自作フォント表示のプログラミング

では、実際のプログラムです。

フォントの読み込み

まず、フォントのグラフィックデータを読み込みます。

ここではシステム側で初期化時に読み込んでいますが、フォントを切り替える場合などは、必要に応じて任意の場所で読み込むと良いでしょう。

```
//フォントテクスチャ(スプライト表示用)の読み込み
D3DXCreateTextureFromFile( g_pD3Ddevice, "..¥¥..¥¥data¥¥font.png",
                           &g_pTex[FONT_BITMAP_ID] );
```

表示座標の取得と改行処理

次に実際の処理関数です。

サンプルは、指定した座標にアルファベットの文字列を表示する関数FontPrintで、引数として表示座標と表示する文字列を必要とします。

LIST 3 −8−1 関数FontPrint

```
#define FONT_SIZE_X 16
#define FONT_SIZE_Y 16

void FontPrint(int x, int y,unsigned char* str)
{
    RECT rect;
    unsigned char alphabet;
    int strx,stry;
    int dispx,dispy;                //表示座標
    D3DXVECTOR3 pos;

    //表示座標を保存
    dispx = x;
    dispy = y;

    while(*str != NULL)
    {//文字列終了までループ
        alphabet = *str++;

        if(alphabet == 0x0a)
        {//改行処理
            dispx = x;
            dispy += FONT_SIZE_Y;
            continue;
        }

        //文字の表示処理
        alphabet -= ' ';
        //元のビットマップ座標を計算
        strx = (alphabet % 0x10) * FONT_SIZE_X;
        stry = (alphabet / 0x10) * FONT_SIZE_Y;
        rect.top    = stry;
```

```
            rect.bottom = stry + FONT_SIZE_Y;
            rect.left   = strx;
            rect.right  = strx + FONT_SIZE_X;

            //表示座標設定
            pos.x = dispx;
            pos.y = dispy;
            pos.z = 0;

            //スプライトで文字表示
            g_pSp->Draw( g_pTex[FONT_BITMAP_ID],
                &rect,
                NULL,
                &pos,
                0xffffffff);

            dispx += FONT_SIZE_X;
        }
    }
```

はじめに、表示する基本の座標をコピーしておきます。

これは、文字列の改行処理で、座標を戻す必要があるためです。

図3 －8－1 フォントデータと、フォントをフォントサイズで分解する図

フォントデータからフォントを切り出してスプライトとして表示

そして文字列から、1文字表示する文字列を読み取ります。

もしこの文字が改行文字（文字コード0x0a）であれば、先ほど保存していた表示元の座標を用いて、改行処理を行ないます。

専用文字コード

次に、フォントを表示する際のデータ座標を表す専用の文字コードを取得します。

専用文字コードとは、各フォントに対して1対1で対応した専用のコードで、使用するフォントの数(ここでは0～63)だけあります。

この時少し工夫をして、フォントデータを通常の文字コードの順番に並べておくと、処理がとても楽になります。

実際、このように配置すると、通常の文字コードから空白文字(文字コード0x20)を引いてやるだけで専用文字コードに変換する事が出来ます。

なお、この変換方法だと、通常文字コード0x20以前の文字で問題が出る事があります。

しかし通常、これらの文字を表示する事はほとんど無いので、問題が出る事は無いでしょう。

フォントの表示

次に、得られた専用文字コードから、表示するフォントのグラフィックデータ上での座標を計算します。

図3－8－2のフォント見てもらえれば分かりますが、文字は16文字単位でならんでいます。

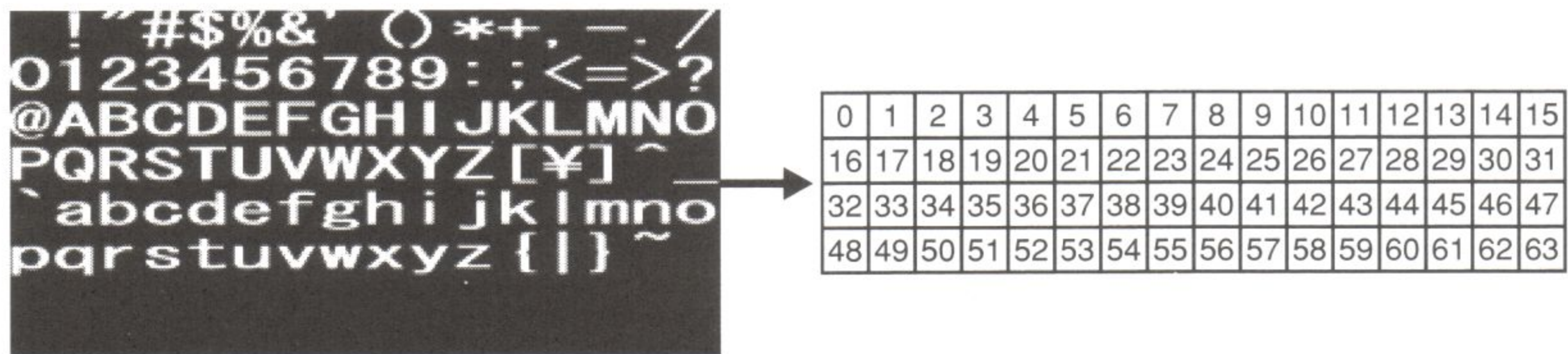
そこで、X座標は16で割った余り、Y座標は16で割った商に、それぞれフォントの大きさ(サンプルでは16ドット)を掛けてやれば、データ上での座標が得られます。

後はこの座標を表示領域とする、スプライトを表示します。座標を矩形領域に設定して、D3DXSPRITEの描画API、Drawを呼び出すだけです。

これを文字列が終了するまで繰り返せば、フォント表示の完成です。

図3－8－2 専用文字コードとフォントデータ座標の概念図

```
 !"#$%&'()*+,-./
0123456789:;<=>?
@ABCDEFGHIJKLMNO
PQRSTUVWXYZ[¥]^_
`abcdefghijklmno
pqrstuvwxyz{|}~
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63

データの並びが文字コード順にしておけば、簡単に変換できる
またフォントは16文字ずつ並んでいるので、文字コードを16で割った商と余りを用いれば、座標を割り出す事ができる

16ドット
16ドット

0	1	2	3
16	17	18	19
32	33	34	35
48	49	50	51

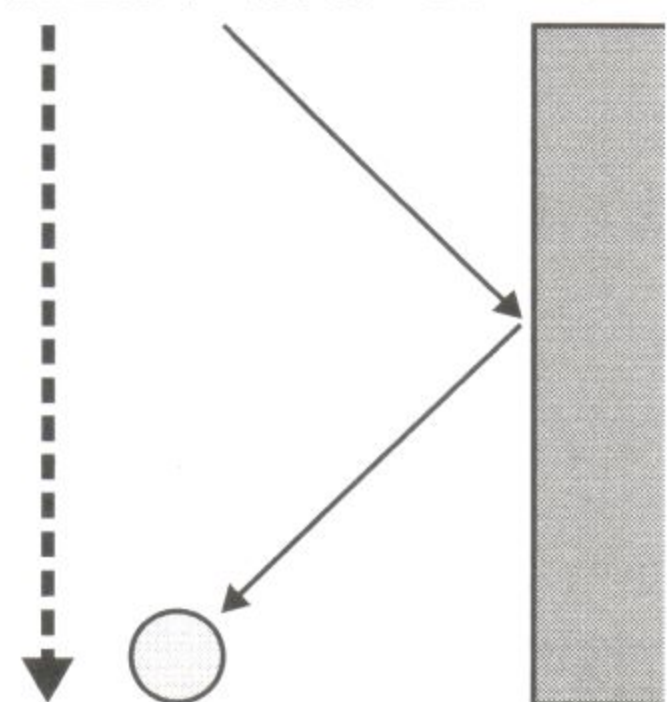
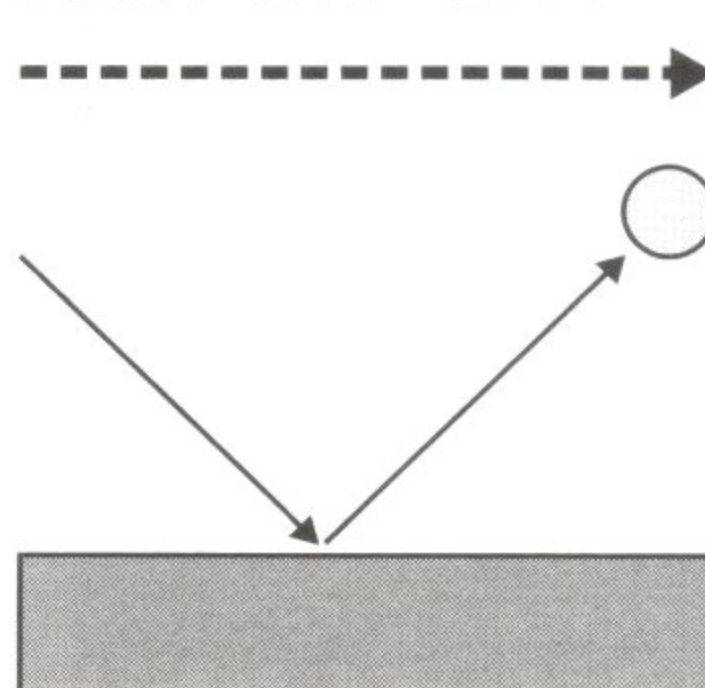
例えば、33の文字コードの座標を得たい時
17÷33で　商が2、余りが1になる
サンプルではフォントの大きさは、16ドットなので商と余り、
それぞれに16を掛けてやれば座標が得られる(X=16、Y=32)
この時、商がY座標、余りがX座標になる

3-9 キャラを壁で跳ね返らせる

跳ね返る処理を考える

キャラを壁で跳ね返らせる処理を作成してみましょう。
この処理は、大きく2つの部分に分けられます。

1つは壁との判定処理です。これは、座標のチェックをX、Y両方の軸に対して行ないます。また、当たり判定の基礎的な処理にもなっています。
もう1つはキャラを跳ね返らせる処理です。これは、図を見てもらえれば分かると思いますが、壁と接触した方向の移動速度を反転させてやる事で実現できます。

図3－9－1 進行方向を反転させる処理のイメージ図

複数のボールが跳ね回るプログラム

初期化部分

ではプログラムを見ていきましょう。サンプルは複数のボールが、画面上を跳ね回る物です。
まず初期化部分ですが、これは本章の最初に説明した、タスクの作成処理の部分です。
タスクの作成後、各タスクの座標と進行方向を設定しています。進行方向の決定は乱数で行なわれ、タスク内の変数、AddX、AddYに設定しています。
初期座標は画面中央になっており、全タスクで共通です。

LIST 3 —9—1 初期化部分のプログラム

```
void exec03_01(TCB* thisTCB)
{

#define ADD_X 8.0
#define ADD_Y 8.0
#define ADD_SPEED 20.0
    TCB* pnew_task;
    TCB* prader_task;
    EX3_1_STRUCT* pnew_work;
    EX3_1_STRUCT* pEX3_1work = (EX3_1_STRUCT*)thisTCB->Work;
    EX3_1_STRUCT* pEX3_rader;

    int loop;
    //レーダー処理の作成
    prader_task = TaskMake(exec03_01_rader, 0x8000);

    //複数表示処理の初期化と起動
    for(loop = 0;loop < EX3_1_MOVE_COUNT; loop++)
    {
        pnew_task = TaskMake(exec03_01_move, 0x2000);
        pnew_work = (EX3_1_STRUCT*)pnew_task->Work;

        //レーダー処理の為に登録
        prader_task->Work[ loop ] = (int)pnew_work;

        //移動量設定
        pnew_work->AddX  =
            (((rand() % 100) / 100.0) - 0.5) * ADD_SPEED;
        pnew_work->AddY  =
            (((rand() % 100) / 100.0) - 0.5) * ADD_SPEED;

        //初期座標
        pnew_work->sprt.X = SCREEN_WIDTH / 2 - 32;
        pnew_work->sprt.Y = SCREEN_HEIGHT / 2 - 32;
    }
    //処理終了
    TaskKill(thisTCB);
}
```

跳ね返るときの処理

次に実際の処理です。

まず、画面の左右に対しての壁の判定を行ないます。

判定後、もしボールの座標が画面端を越えていたら、すなわち壁に接触していたら、進行方向のX方向の移動値を反転させます。

移動値を反転させる事により、その壁に対してボールが反射しているように見えます。

本来は、左右の壁別々でチェックするものですが、反転させる処理自体は両方の壁で同じため、左右の壁を同時にチェックして、処理を簡略化しています。

次に上下の壁の処理ですが、チェックする壁の方向と、反転させる移動値がＹ方向の物に変わるだけで、処理内容自体はまったく同じです。

最後に、移動値を座標に対して加算してやり、ボールを表示してやれば処理は終了です。

LIST 3 －9－2 跳ね返す部分のプログラム

```
void exec03_01_move(TCB* thisTCB)
{
#define WALL_TOP     0         //壁上座標
#define WALL_BOTTOM 480        //壁下座標
#define WALL_LEFT    0         //壁右座標
#define WALL_RIGHT  640        //壁左座標

    EX3_1_STRUCT* pEX3_1work;
    pEX3_1work = (EX3_1_STRUCT*)thisTCB->Work;


    //もし、進行した先が壁であれば方向を反転させる
    if( (pEX3_1work->sprt.X+32 + pEX3_1work->AddX >= WALL_RIGHT) ||
//＋32は表示物の大きさを考慮
        (pEX3_1work->sprt.X    + pEX3_1work->AddX <= WALL_LEFT ) )
    {
        pEX3_1work->AddX = -pEX3_1work->AddX;
    }
    if( (pEX3_1work->sprt.Y+32 + pEX3_1work->AddY >= WALL_BOTTOM) ||
        (pEX3_1work->sprt.Y    + pEX3_1work->AddY <= WALL_TOP    ) )
    {
        pEX3_1work->AddY = -pEX3_1work->AddY;
    }
```

```
    pEX3_1work->sprt.X += pEX3_1work->AddX;
    pEX3_1work->sprt.Y += pEX3_1work->AddY;

    SpriteDraw(&pEX3_1work->sprt,0);
}
```

3-10 パッケージ化

データをまとめる理由

市販のゲームでは、使用するデータを１つのファイルにまとめている物が良くあります。
もちろん、このようにまとめるのには理由があります。

まず、使用するデータをゲーム以外で使用されないようにするため。
そして、細かいファイル作成によるHDDの無駄な使用量を節約するためです。
この点に関してはピンと来ない方もいるかもしれませんが、細かいファイルを沢山作るとそれだけでHDDの容量を消費してしまうのです。

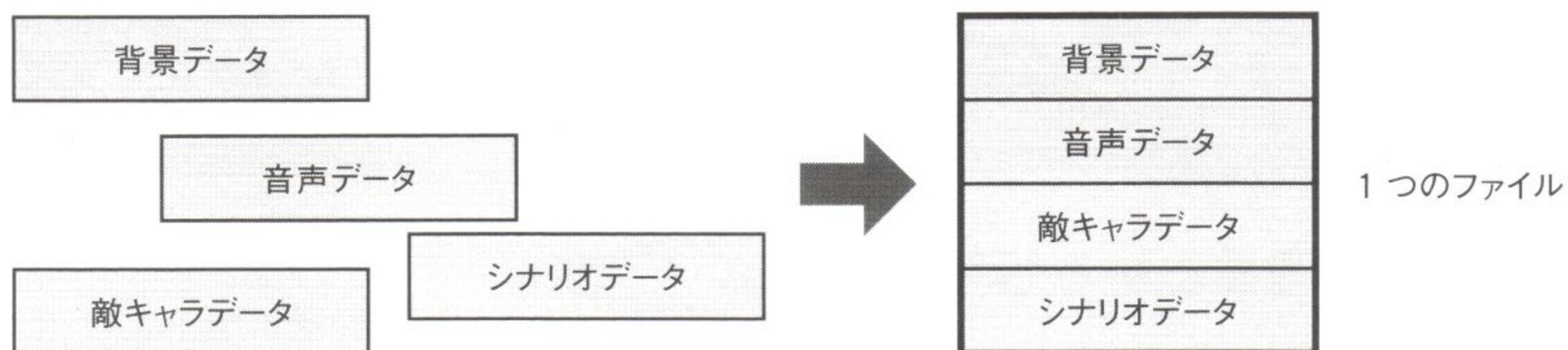
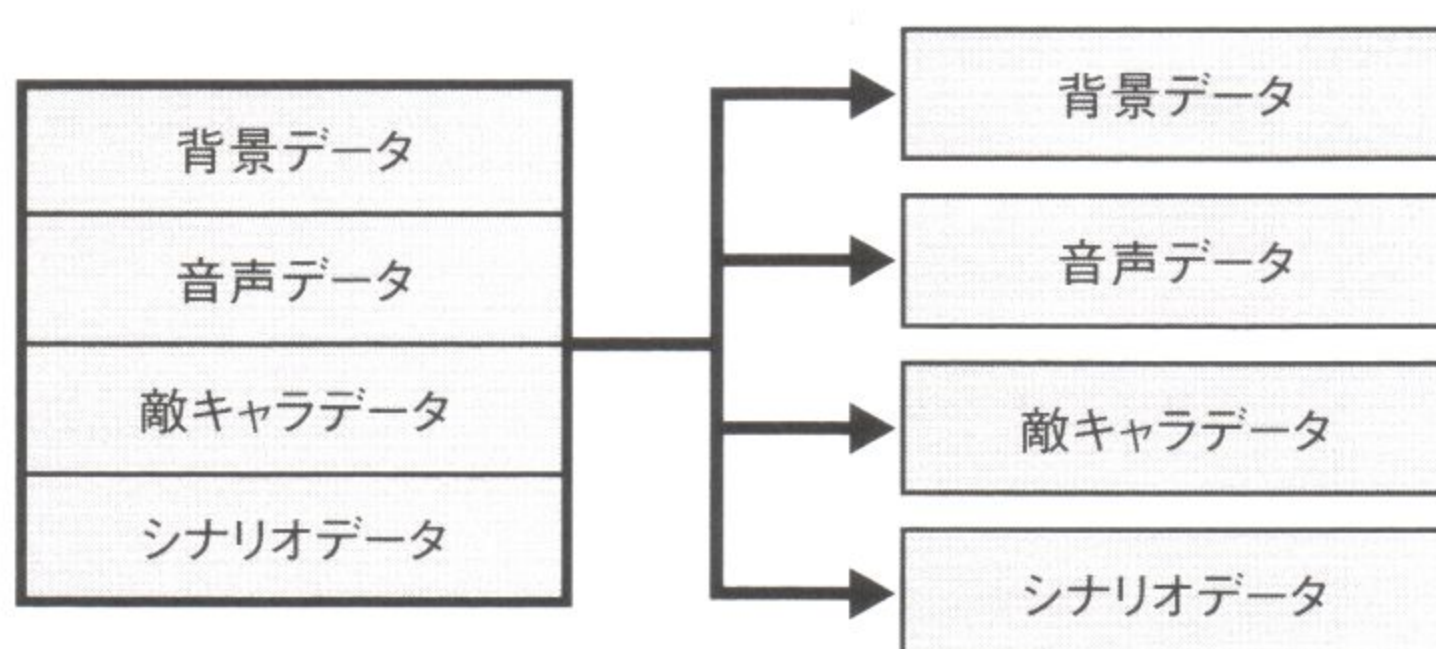
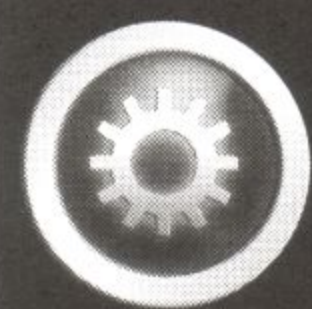
そのため、細かいデータをなるべく大きな１つのデータにする必要が出てきます。
このような複数のデータを１つのファイルにまとめる事を「アーカイブ」または「パッケージング」と呼んでいます。

パッケージ化に関する処理内容

このパッケージ化の処理は、大きく２つの処理に分けられます。
１つは複数のデータを１つのファイルにまとめる処理です。
これは、通常ゲームの開発時にのみ行なわれますが、稀にゲームのセーブデータをまとめる際にも使用されます。

そしてもう１つは、まとめられたファイルにアクセスする処理です。
これはゲームのプログラム側で行なわれる処理で、ゲーム中必要なデータを取得する際に使用されます。

なお、この処理は性質上、使用頻度が高くなるので、なるべく使いやすいようにする必要があります。

図3 ―10―1 まとめる処理と、データを取得する処理のイメージ図

パッケージングはバラバラのデータを1 つにファイルにまとめる処理

ゲーム中で使用するには1つのファイルから必要なデータを抽出する処理が必要

3-11 データをまとめるプログラム

データをまとめる処理

では実際に処理の作成を行なってみましょう。
まずは、データをまとめる（アーカイブ）の処理プログラムからです。

処理の流れですが、関数 EX03_11_archive_data を呼び出す事で一気に処理を行ないます。
その為に、この関数は、2つの引数を受け取ります。

1つ目の第一引数は、まとめるファイルのあるフォルダを指定するものです。ここで指定した（HDD内の）フォルダ内にある全てのファイルをまとめます。
この際に対象となるファイルは、バイナリ・テキスト等は関係なく、全てのファイルが対象になります。
次に2つ目、第二引数は、第一引数で指定したファイルをまとめる事で、新たに作られるファイルの名前を指定します。まとめる事で出来るファイルは1つですので、当然指定のファイル名も1つになります。
この時作成されるファイルのデータフォーマットは以下の様になっています。

図3－11－1 データフォーマットの図

型	名前	内容	備考
unsigned int	FileCount	格納されているファイル数	パッケージファイルに1つだけ存在する
unsigned int	FilePoint	格納されているファイル位置	FILE_MAX 分だけ並ぶ(サンプルでは32 個)
unsigned int	FileSize	格納されているファイルサイズ	
TCHAR	FileName	格納されているファイル名	
⋮			
実際のデータ			

なお、アーカイブできるファイル数は上限があり、その数はマクロで定義されています。この時、定義された数に合わせてヘッダ部分のサイズは固定になります。

ちなみに、作成されるファイルは専用フォーマットの為、元のファイルがどのような物であっても、Windowsからは1つのバイナリファイルとして扱われます。

あと、ここでいう「アーカイブ」とは、ファイルをまとめる処理を指します。一般的なアーカイブソフト(LZHやZIP)と違って、まとめるだけで、ファイルに圧縮や暗号化を行なう事はありません。

もし暗号化を行なうのであれば、[3－22]を参考にしてみてください。

さて次は、実際の関数の内部を解説していきます。ですが、出てくる変数名が多く、少々ややこしくなっていますので注意してください。

フォルダ内のファイルを調べる

まずは、指定したフォルダの中にあるファイルを調べます。

最初に、指定したフォルダ内のファイルを調べるため、ワイルドカードを検索フォルダの文字列に追加します。

そして実際にフォルダ内のファイルを調べるにはWindowsのAPI、FindFirstFile、FindNextFile、FindCloseの3つの関数を使います。

ファイルを調べる

まず、FindFirstFileで指定した条件のファイル名でフォルダを検索し、FindNextFileで条件に合致したファイルの情報を次々と取得します。

ここでは、条件にワイルドカードを指定しているので、全てのファイルが対象になります。

この際、FindNextFileで最初に取得するファイルはフォルダの情報なので、とばします。この時同時にファイル数もカウントしますが、この数は含まれません。

調べたファイルの情報を保存し、終了

取得した情報は、ファイルの先頭部分に書き込むヘッダ情報として、配列変数file_headerに保存しておきます。

これをフォルダ内のファイルが無くなるまで続け、処理が終了したらFindCloseを呼び出します。

この時、ファイルが無くなったかどうかはFindNextFileがエラーを起こした時に、関数GetLastErrorがERROR_NO_MORE_FILESを返すかどうかでチェックできます。

ファイルをまとめる

ファイルの情報を取得する

フォルダ内のファイルを調べ終わったら、次は実際のファイルをまとめる処理です。

はじめに、ファイルの情報を記録したヘッダ部分を記録します。

ヘッダ部分は、先に紹介したフォーマット情報にもあるように、ファイル数とファイル情報部分から出来ています。

ファイル数は先ほどカウントしておいた数値を、ファイル情報部分は配列変数 file_header をそのまま記録します。

ひとつにまとめる

次にフォルダ内部のファイルを、今記録したヘッダー部分に続けてコピーします。

ファイル名はヘッダ情報部分から取得、指定したフォルダ名と合成してコピーファイル名を取得しています。

後はこれを、ファイル数分だけ繰り返せば、処理は終了です。

LIST 3 －11－1 データをまとめる

```
typedef struct{
	//先頭からのファイルの位置
	unsigned int	FilePoint;
	//ファイルサイズ
	unsigned int	FileSize;
	//ファイル名
	TCHAR		FileName[ MAX_PATH ];
} EX03_11_DATA;
#define FILE_MAX	32
#define HEADER_SIZE (sizeof(EX03_11_DATA) * FILE_MAX + sizeof(int))

int EX03_11_archive_data( char* DirName, char* SaveName)
{
	int file_count = 0;	//ファイル数
	WIN32_FIND_DATA FindFileData;
	HANDLE hFind;
	//ファイル先頭からの位置
	unsigned int file_point = HEADER_SIZE;

	FILE* archive_file;
```

```
    FILE* read_file;
    //検索するファイル名
    char search_name[MAX_PATH];
    char new_name[MAX_PATH];
    unsigned int copy_byte;
    int loop;
    unsigned int copy_loop;

    //アーカイブファイルのヘッダ部分
static EX03_11_DATA file_header[FILE_MAX];


    //検索するファイル名を作成
    //フォルダ内の全ファイルを指定する
    strcpy(search_name,DirName);
    strcat(search_name,"*.*");

    //最初の1度だけ実行
    hFind = FindFirstFile( search_name, &FindFileData);
    //最初のファイルはディレクトリなのではずす
    FindNextFile( hFind, &FindFileData);
    while(true)
    {
        //ファイルを検索
        if( FindNextFile( hFind, &FindFileData) == 0)
        {//エラーチェック
            if( GetLastError () == ERROR_NO_MORE_FILES )
            {//ファイルがもうなければ検索ループを終了
                break;
            }else{
              //そうでなければエラーで戻る
                FindClose(hFind);
                return true ;
            }
        }

        //ヘッダに情報を書き込む
        file_header[ file_count ].FilePoint =
            file_point;
        file_header[ file_count ].FileSize  =
```

```
                    FindFileData.nFileSizeLow;

                //ファイル名のコピー
                strcpy( file_header[ file_count ].FileName,
                    FindFileData.cFileName);

                //格納するファイルの先頭からの位置計算
                file_point += FindFileData.nFileSizeLow;

                //ファイル数をカウントする
                file_count++;
            }
            //検索終了
            FindClose(hFind);

            //バイナリ書き込みモードでまとめるファイルを開く
            archive_file = fopen( SaveName, "w+b");

            //最初にヘッダ部分を記録する
            //ファイル数
            fwrite( &file_count, sizeof(int) , 1, archive_file );
            //ヘッダ本体(ファイル数を記録した分減らす)
            fwrite( file_header, HEADER_SIZE-sizeof(int) , 1, archive_file );

            for(loop=0; loop < file_count; loop++ )
            {//個別のファイルを1つのファイルにコピーする
                strcpy(new_name,DirName);
                strcat(new_name, file_header[loop].FileName);
                read_file = fopen(new_name, "r+b");

                for( copy_loop = 0; copy_loop < file_header[loop].FileSize;
copy_loop++)
                {//ファイルサイズ分だけ繰り返す
                    fread(  &copy_byte, 1, 1, read_file );
                    fwrite( &copy_byte, 1, 1, archive_file );
                }

                fclose( read_file );
            }
```

```
    //ファイルを閉じる
    fclose( archive_file );

    //正常終了
    return false;
}
```

3-12 まとめたデータにアクセスする

まとめたデータにアクセスする処理

最後に、まとめたデータにアクセスする処理を作成します。

この処理は、2つの段階に分かれています。

1つは、まとめたファイルを内部に読み込む処理、もう1つは読み込んだファイルからデータを取得する処理です。

前者の処理は、一般的なファイルの読み込み処理ですので問題はありません。ただ、後者は一連の処理を行なう関数群になりますので、使用方法が若干分かりにくくなります。

そのためサンプル処理として、関数を使用して取得した情報を用い、まとめたファイルを元のバラバラのファイルとして復元する処理を行なっています。

データにアクセスするプログラム

まずは、ファイルの読み込み処理です。ファイルのサイズを取得し、全データを読み込んでいます。

その後、データを取得する以下の関数を用いて、ファイルを書き込んでいます。

EX03_11_getFileCount:　**まとめられたファイルの数を取得します。**
EX03_11_getFileName:　**指定したファイルIDのファイル名を取得します。**
EX03_11_getFilePoint:　**指定したファイルIDのファイルへのポインタを取得します。**
EX03_11_getFileSize:　**指定したファイルIDのファイルサイズを取得します。**

ファイルIDとは、各ファイルに対応付けられた一連の番号の事です。

これらの情報を用いて読み込んだファイルからファイル情報を取得し、復元処理を行なっています。

ヘッダ情報に大半の情報が書き込まれているため、各関数の中身は非常にシンプルです。

なお、ヘッダ情報にはファイル名も含まれているため、IDではなくファイル名で検索する事も難しくはないでしょう。

LIST 3 —12—1

```
int EX03_11_getFileCount( char* Buff )
{
	//先頭にファイル数が格納されている
	return  *((int*)Buff);
}

unsigned int EX03_11_getFileSize( char* Buff ,int ID)
{
	//データ配列の先頭をポイントする
	EX03_11_DATA* data = (EX03_11_DATA*)(Buff + sizeof(int) );
	//ファイルサイズを返す
	return data[ID].FileSize;
}

char* EX03_11_getFileName( char* Buff ,int ID)
{
	//データ配列の先頭をポイントする
	EX03_11_DATA* data = (EX03_11_DATA*)(Buff + sizeof(int) );
	//ファイル名へのポインタを返す
	return data[ID].FileName;
}

char* EX03_11_getFilePoint( char* Buff ,int ID)
{
	//データ配列の先頭をポイントする
	EX03_11_DATA* data = (EX03_11_DATA*)(Buff + sizeof(int) );
	//バッファポインタからファイル格納位置へのポインタを算出
	return (char*)(data[ID].FilePoint+(int)Buff);
}



void exec03_11(TCB* thisTCB)
{
	//ファイルの読み込みバッファ(32K)
	char read_buff[1024*32];
	int file_count;
	char* file_name;
	int   file_size;
```

```
char* file_point;
int loop;
int archive_size;
FILE* archive_file;
FILE* write_file;

//読み込み処理
archive_file = fopen( "..¥¥..¥¥data¥¥EX03_11.DAT","r+b");
fseek( archive_file, 0, SEEK_END );
//ファイルサイズを取得
archive_size = ftell( archive_file);
fseek( archive_file, 0, SEEK_SET );
//ファイルを全部読み込む
fread( &read_buff, 1, archive_size, archive_file );
//読み込み終了後ファイルを閉じる
fclose( archive_file );


//ファイル数の取得
file_count = EX03_11_getFileCount(read_buff);
for(loop = 0; loop< file_count; loop++)
{//まとたファイルをバラバラに書き込む
    //書き込みに必要な各種情報を取得
    //ファイル名
    file_name  = EX03_11_getFileName (read_buff,loop);
    //ファイルサイズ
    file_size  = EX03_11_getFileSize (read_buff,loop);
    //格納されているファイルへのポインタ
    file_point = EX03_11_getFilePoint(read_buff,loop);

    write_file = fopen(file_name ,"w+b");
    fwrite( file_point, file_size, 1, write_file);
    fclose( write_file );
}

//処理の終了
TaskKill(thisTCB);
```

3-13 ポーズをかける

ポーズ機能を考える

ゲームをプレイ中、ちょっと一息入れたい時など、ゲームを一時的に停止してくれるポーズ機能はなくてはならない物です。

このポーズ機能は、環境によってはハードウェアで実装している場合もありますが、大半はソフトウェアによる処理になります。

単純に処理を止めるだけでは不十分

では、このポーズの処理ですが、具体的にどんな事をしているのでしょうか？

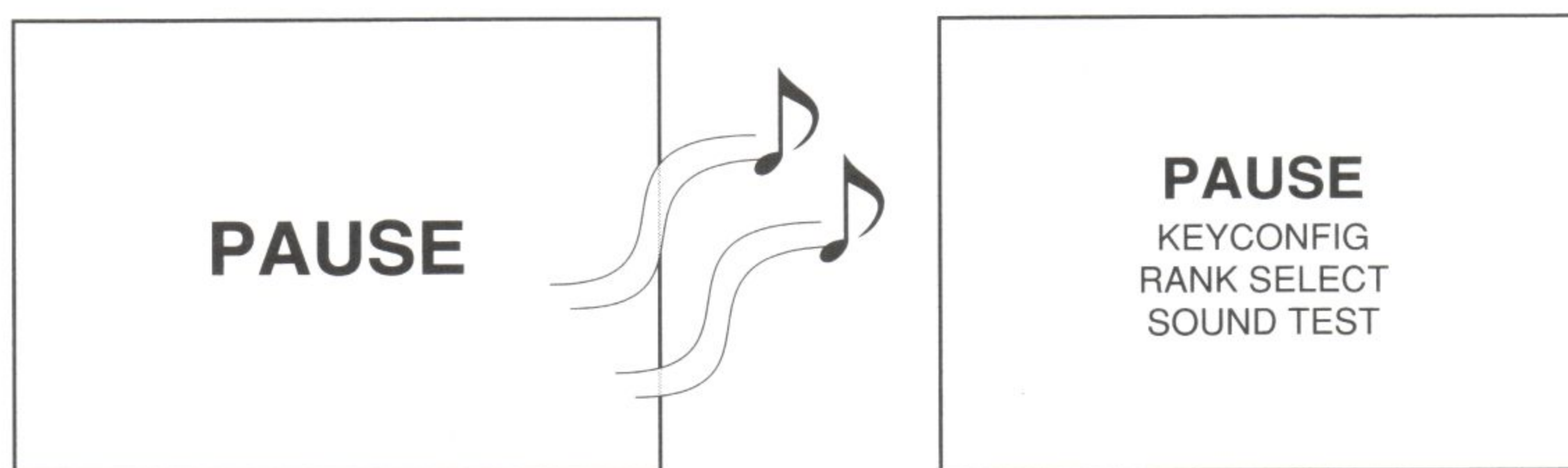
単純に考えれば、処理を全部止めればいいような気がしますが、処理そのものを停止してしまうと、ゲーム画面の表示処理そのものも行なわれなくなり、画面に何も表示されなくなってしまいます。

また仮に、ゲーム画面だけ残すようにしたとしても、ポーズ中に、その他の事が処理できなくなってしまい、いささか不便になってしまいます。

例えば、サウンド再生の管理を行なったり、ポーズ中である事を表示したり、場合によっては時間を表示したり等、ポーズ中でも処理を続けたい場合があるからです。

図3 －13－1 ポーズ中でも処理をしていないと不便な例のイメージ図

PAUSE

PAUSE
KEYCONFIG
RANK SELECT
SOUND TEST

ポーズ中でも曲を流したり、オプションの設定を行なうなど動作していないと不便な事がある

どのような処理をすればいいか

　では、どうすればいいのかというと、発想を変えて、ポーズ中に動いている処理があるのではなく「ポーズをしたら止まる処理がある」と考えるのです。
　具体的には、ポーズの状態を管理するポーズフラグを設けて、ポーズをさせたい処理だけ停止をすればよいのです。
　そうすればポーズ中であろうと、自由に色々な処理を行なう事が出来ます。

ポーズ機能のプログラム

　それでは、サンプルプログラムを見ていきましょう。
　サンプルは、上から多数のボールが落ちてくるプログラムです。

初期化

　まずは初期化です。ボールの作成と座標の初期化を行なっています。
　初期化の最後にポーズのフラグを初期化します。
　ポーズフラグはタスク同士での通信を用いてもいいのですが、多様な処理から参照される事を考えて、グローバル変数を用いています。

メイン処理

　次にメインの処理ですが、ここはポーズのフラグを管理しているだけです。
　ボタンを押すたびにポーズと、ポーズの解除を行ないます。

ポーズの処理

　次は、実際のポーズを行なうボールの処理です。
　ポーズ処理自体は非常に単純で、ポーズフラグを監視しておき、ポーズ状態の時には移動に関する処理を行なわないようにします。

　この時注意する事は、ポーズ中でも表示の処理をきちんと行なうようにする事です。
　サンプルは比較的単純な例ですが、処理が複雑になってくると、ゲームの処理と表示の処理を分けるのが、面倒になる場合があります。

　1つの関数内で、複数の異なる物体の表示を行なっている場合等がそうで、1つの表示毎にポーズフラグを設けて、ポーズの処理を行なわなくてはなりません。
　そのような場合に対処する為にも、ゲーム処理と表示処理をキチンと分けるようにして下さい。
　管理が難しいようなら処理システムを改良して、管理しやすいようにしても良いでしょう。

LIST 3 －13－1 ポーズをかける

```
typedef struct{
    SPRITE          sprt;
    SPRITE*         Target;                     //目標のスプライト
} EX03_13_STRUCT;

//ポーズフラグ
int g_EX03_13_PAUSE_FLAG;

void exec03_13_Ball(TCB* thisTCB)
{
#define MOVE_SPEED  8.0
    SPRITE* work = (SPRITE*)thisTCB->Work;

    //ボールの移動処理部分 ポーズ中には処理しない
    if( !g_EX03_13_PAUSE_FLAG )
    {
        work->Y += MOVE_SPEED;
        //画面外に到達したら座標を戻す
        if( work->Y  > SCREEN_HEIGHT ) work->Y = 0;
    }

    //描画部分はポーズ中でも常に描画し続ける
    SpriteDraw(work,0);
}

void init03_13(TCB* thisTCB)
{
#define BALL_COUNT 32
    TCB*    tmp_tcb;
    SPRITE* ball_sprt;
    int loop;


    //使用するテクスチャの読み込み
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥0055.png",&g_pTex[0] );

    //ボールの作成と座標の初期化
    for( loop = 0; loop < BALL_COUNT; loop++)
```

```
        {
            tmp_tcb = TaskMake( exec03_13_Ball, 0x1000 );
            ball_sprt = (SPRITE*)tmp_tcb->Work;

            ball_sprt->X = rand() / (RAND_MAX / SCREEN_WIDTH);
            ball_sprt->Y = rand() / (RAND_MAX / SCREEN_HEIGHT);
        }
        //ポーズフラグの初期化
        g_EX03_13_PAUSE_FLAG = 0;
}

void exec03_13(TCB* thisTCB)
{
        //ポーズフラグの処理
        if( g_DownInputBuff & KEY_Z )
        {//ボタンを押すたびにフラグを反転させ、ポーズとポーズ解除を行う
            g_EX03_13_PAUSE_FLAG = !g_EX03_13_PAUSE_FLAG;
        }
}
```

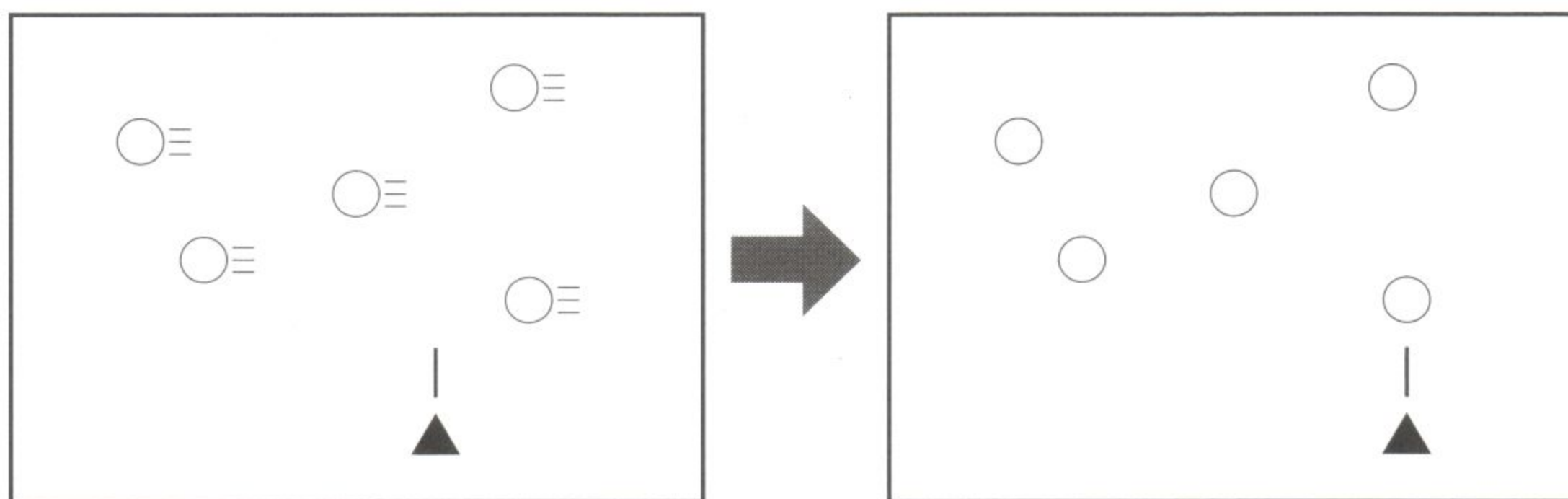
3-14 グループ別にポーズをかける

特定のグループの処理を止める

次はグループ別にポーズをかける様にしてみましょう。

一般的なポーズ処理では、ゲーム全体を停止させますが、ここでのポーズ処理は、主にゲーム中で使用されるものです。

例えば、ゲーム中のアイテム効果で一部の敵だけ止めたり、画面エフェクト等で画面は一応停止しているが、文字や爆発は表示させたりしている事などです。

図3 －14－1 ゲーム中、特定のグループだけ処理を止めるのイメージ図

動き回って当てにくい敵を、アイテムを取って停止させる・・・等、特定のグループを止めたい処理は意外と多い

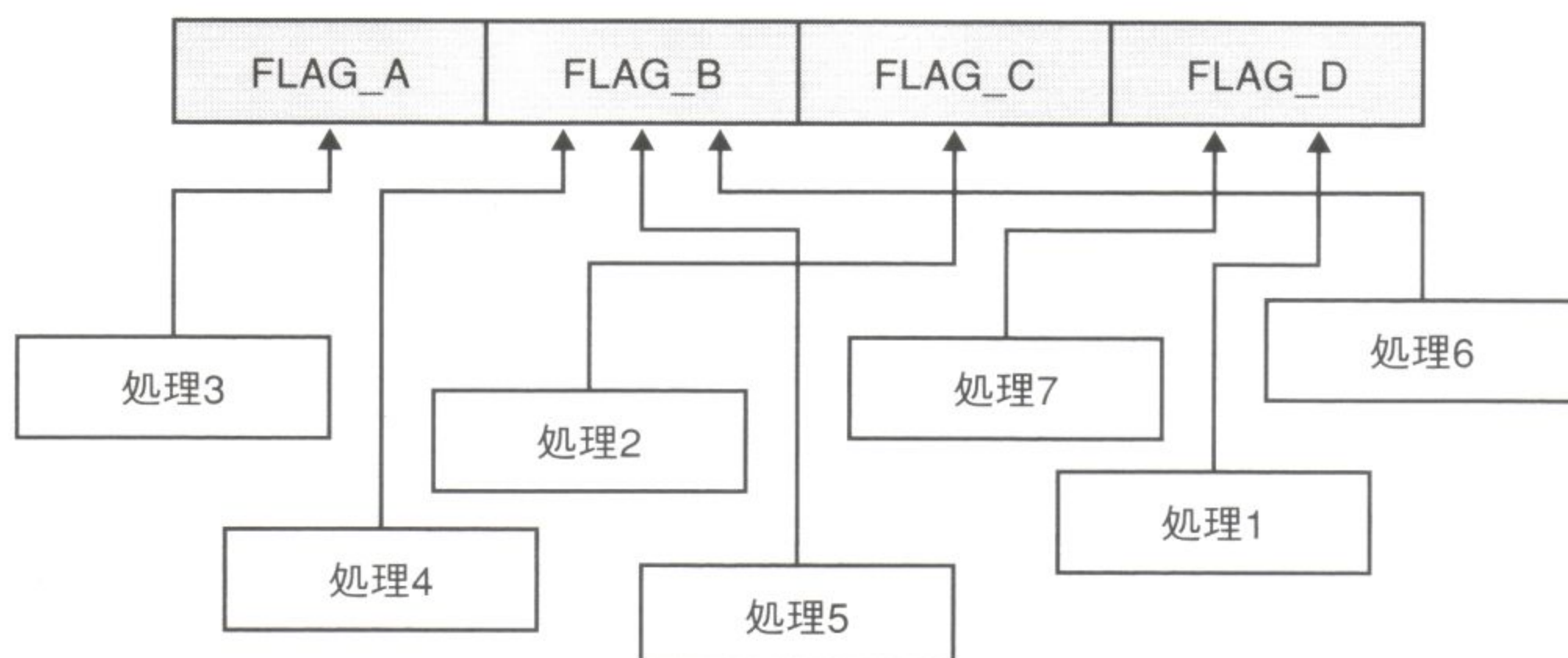
どのように処理するか

処理の概要ですが、グループ別といってもID等を設けて、グループを分ける訳ではありません。

ポーズフラグを細かく分け、処理する側がどのフラグを参照してポーズを行なうかで、グループ分けを行ないます。

言い換えれば、どのフラグを参照するかという事で、ポーズのグループが決定されます。

こうする事で、積極的な管理を行なわずに、少ない情報で全体を管理できます。

図3 －14－2 細かいフラグで、グループを管理するイメージ図

各処理が割り当てられたフラグをチェックする事で、グループ分けを行なう

特定グループの動きを止めるプログラム

では、サンプルプログラムを見ていきましょう。

サンプルは上下左右に流れているボールを、対応した方向キーを押す事により、ポーズ・ポーズ解除します。

初期化

最初に、初期化ですが、ボールの作成と座標の初期化を行なっています。

ボールは作成時に、上下左右に流れる４つの種類に分けて作成しており、その後ポーズフラグを初期化します。

このフラグも、通常のポーズフラグと同様に多数の処理から参照されるため、グローバル変数を用いています。

ただ、中身は少し扱い方が変わり、上下左右の各方向に対応した４つのポーズフラグを持つようになっています。

メイン処理

次にメインの処理ですが、キー入力に応じたフラグの状態管理を行なっています。

各方向キーが押されるたびに、方向に対応したポーズフラグを反転させ、ポーズとポーズの解除を行ないます。

ボールの処理

最後にボールの処理です。

上下左右に応じた４つの処理がありますが、それぞれ参照するポーズフラグが違っています。

説明したように、参照されるポーズフラグはそれぞれ独立しているため、特定の処理だけポーズ

をかける事が可能です。

参照してポーズ状態なら、移動の処理を行なわないようにします。

もちろん、行なわないのは移動処理だけで、表示の処理をする事を忘れないで下さい。

LIST 3 −14−1 グループ別にポーズをかける

```
//ポーズフラグ
#define PAUSE_GROUP_A  (1 << 0)
#define PAUSE_GROUP_B  (1 << 1)
#define PAUSE_GROUP_C  (1 << 2)
#define PAUSE_GROUP_D  (1 << 3)
int g_EX03_14_PAUSE_FLAG;

//ボール種類A
void exec03_14_BallA(TCB* thisTCB)
{
#define MOVE_SPEED  8.0
    SPRITE* work = (SPRITE*)thisTCB->Work;

    //ボールの移動処理部分 ポーズ中には処理しない
    if( !(g_EX03_14_PAUSE_FLAG & PAUSE_GROUP_A) )
    {
        work->Y += MOVE_SPEED;
        //画面外に到達したら座標を戻す
        if( work->Y  > SCREEN_HEIGHT ) work->Y = 0;
    }
    //描画部分はポーズ中でも常に描画し続ける
    SpriteDraw(work,0);
}

//ボール種類B
void exec03_14_BallB(TCB* thisTCB)
{
#define MOVE_SPEED  8.0
    SPRITE* work = (SPRITE*)thisTCB->Work;

    //ボールの移動処理部分 ポーズ中には処理しない
    if( !(g_EX03_14_PAUSE_FLAG & PAUSE_GROUP_B) )
    {
        work->Y -= MOVE_SPEED;
```

```
            //画面外に到達したら座標を戻す
            if( work->Y  < 0 ) work->Y = SCREEN_HEIGHT;
        }
        //描画部分はポーズ中でも常に描画し続ける
        SpriteDraw(work,0);
}

//ボール種類C
void exec03_14_BallC(TCB* thisTCB)
{
#define MOVE_SPEED  8.0
        SPRITE* work = (SPRITE*)thisTCB->Work;

        //ボールの移動処理部分　ポーズ中には処理しない
        if( !(g_EX03_14_PAUSE_FLAG & PAUSE_GROUP_C) )
        {
            work->X += MOVE_SPEED;
            //画面外に到達したら座標を戻す
            if( work->X  > SCREEN_WIDTH ) work->X = 0;
        }
        //描画部分はポーズ中でも常に描画し続ける
        SpriteDraw(work,0);
}

//ボール種類D
void exec03_14_BallD(TCB* thisTCB)
{
#define MOVE_SPEED  8.0
        SPRITE* work = (SPRITE*)thisTCB->Work;

        //ボールの移動処理部分　ポーズ中には処理しない
        if( !(g_EX03_14_PAUSE_FLAG & PAUSE_GROUP_D) )
        {
            work->X -= MOVE_SPEED;
            //画面外に到達したら座標を戻す
            if( work->X  < 0 ) work->X = SCREEN_WIDTH;
        }
        //描画部分はポーズ中でも常に描画し続ける
        SpriteDraw(work,0);
}
```

```
void init03_14(TCB* thisTCB)
{
#define BALL_COUNT 32
    TCB*    tmp_tcb;
    SPRITE* ball_sprt;
    int loop;


    //使用するテクスチャの読み込み
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥0055.png",&g_pTex[0] );

    //ボールの作成と座標の初期化
    for( loop = 0; loop < BALL_COUNT; loop++)
    {
        switch( loop & 0x03 )    //ボールの種類を4つに分ける
        {
            case 0:  tmp_tcb = TaskMake( exec03_14_BallA, 0x1000 ); break;
            case 1:  tmp_tcb = TaskMake( exec03_14_BallB, 0x1000 ); break;
            case 2:  tmp_tcb = TaskMake( exec03_14_BallC, 0x1000 ); break;
            case 3:  tmp_tcb = TaskMake( exec03_14_BallD, 0x1000 ); break;
        }

        ball_sprt = (SPRITE*)tmp_tcb->Work;

        ball_sprt->X = rand() / (RAND_MAX / SCREEN_WIDTH);
        ball_sprt->Y = rand() / (RAND_MAX / SCREEN_HEIGHT);
    }
    //ポーズフラグの初期化
    g_EX03_14_PAUSE_FLAG = 0;
}

void exec03_14(TCB* thisTCB)
{
    //ポーズフラグの処理、方向キーに合わせてフラグ反転処理
    //各キーを押すたびに、ポーズとポーズ解除を行う

    if( g_DownInputBuff & KEY_UP )
    {//上
```

```
        g_EX03_14_PAUSE_FLAG ^= PAUSE_GROUP_A;
    }

    if( g_DownInputBuff & KEY_DOWN )
    {//下
        g_EX03_14_PAUSE_FLAG ^= PAUSE_GROUP_B;
    }

    if( g_DownInputBuff & KEY_LEFT )
    {//左
        g_EX03_14_PAUSE_FLAG ^= PAUSE_GROUP_C;
    }

    if( g_DownInputBuff & KEY_RIGHT )
    {//右
        g_EX03_14_PAUSE_FLAG ^= PAUSE_GROUP_D;
    }
}
```

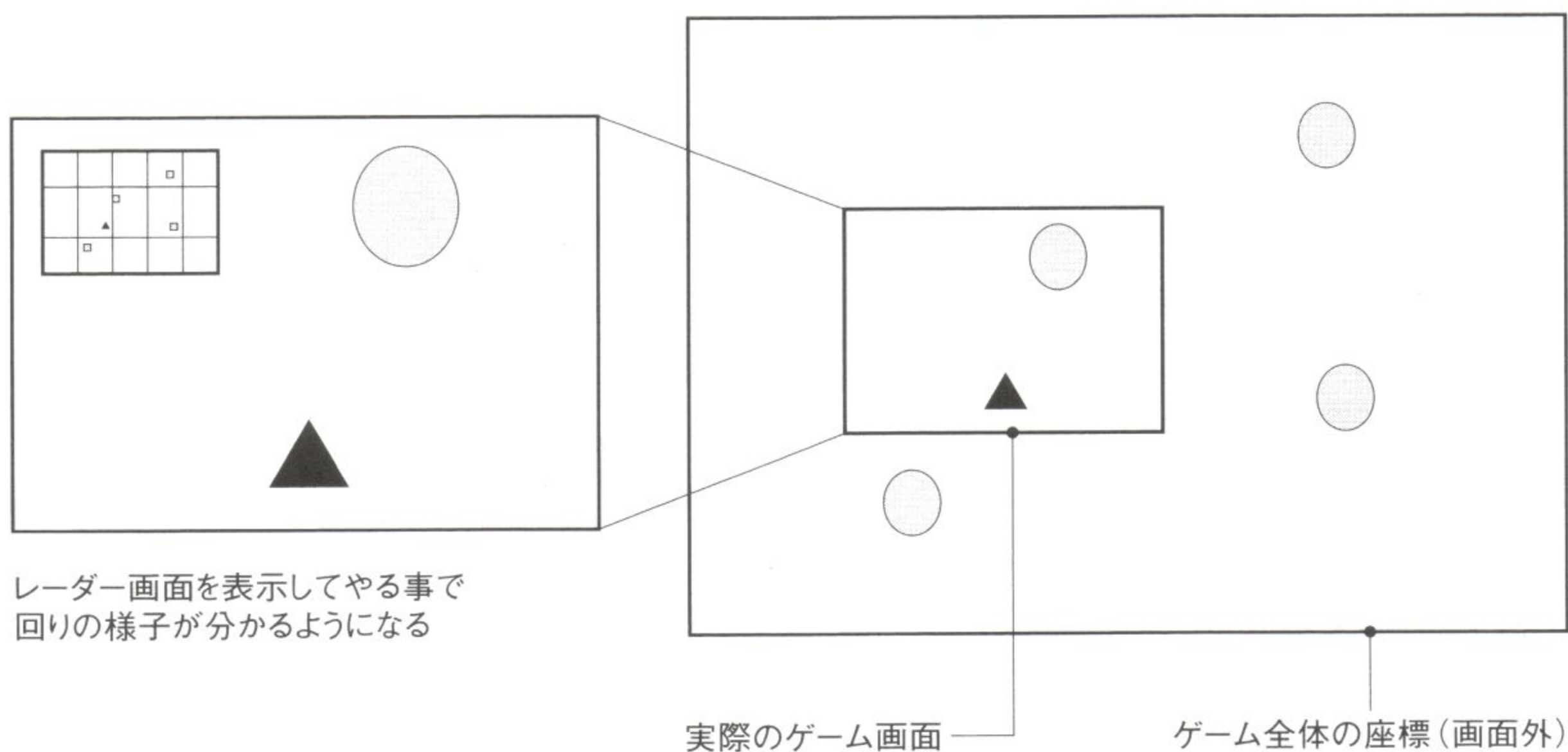
3-15 レーダー画面の処理

ゲーム中のレーダー画面

レーダー画面とは、ゲーム中のキャラクターの座標を小さな画面で表示する機能の事です。

スクロールするゲームで敵や味方の状況を探ったり、位置を確認して戦略を練ったりする場合などに使用されます。

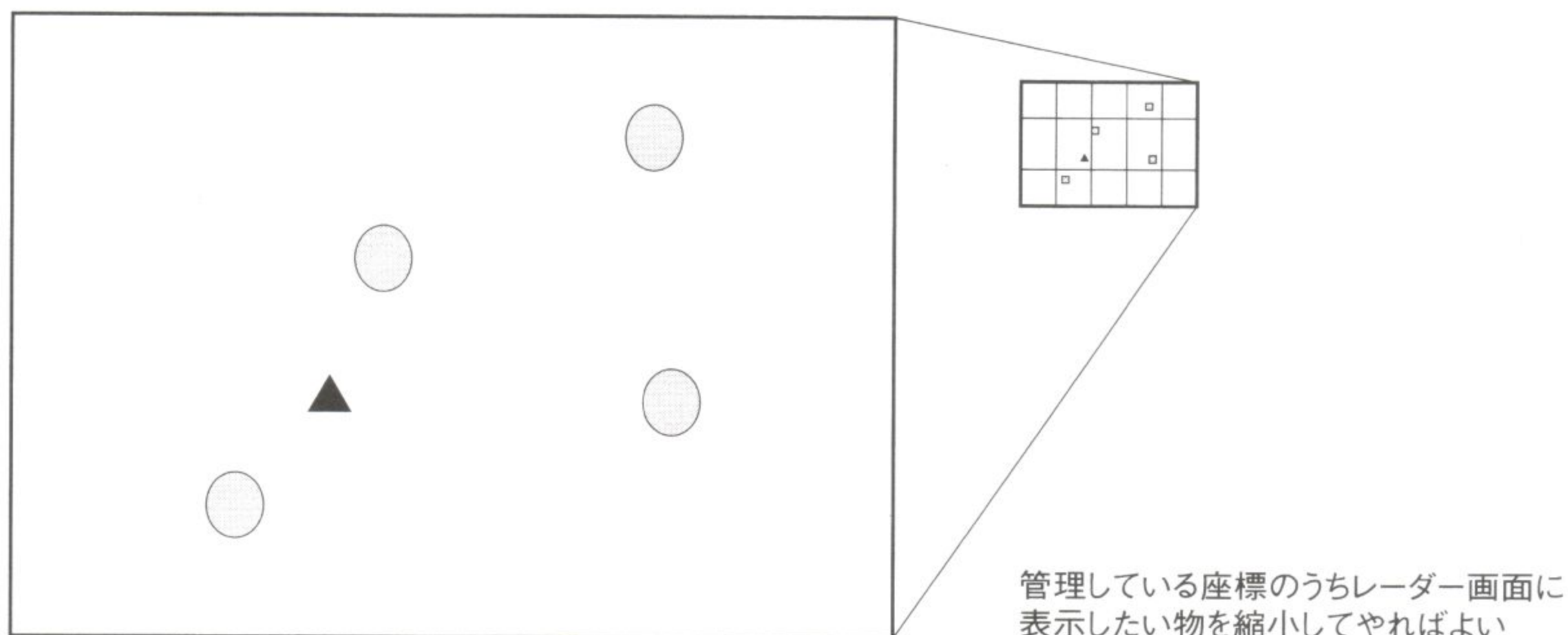
画面の内外の情報を一瞥して判断できるので、ゲーム内容によっては必須とも言える機能でしょう。

図3 －15－1 ゲーム中でのレーダー画面のイメージ図

どのように処理するか

処理の概要ですが、基本的に、レーダーに表示をさせたい全てのキャラクターの座標を管理する必要があります。

そして、その座標を表示座標から、レーダー画面の座標に変更して表示してやります。

要するに、通常表示の座標を使用して、レーダー画面にキャラクターを表示するのです。

図3 －15－1 レーダー画面での座標変更のイメージ図

レーダー画面のプログラム

では、実際のプログラムを見ていきましょう。

サンプルは、[3－9]で紹介した、画面上を跳ね回るボールのキャラクターを、画面左上にレーダー画面として表示する物です。

初期化

まず初期化ですが、まず表示するキャラクター座標を管理するための処理を作成します。

これは、本章の最初で紹介したプログラムから、タスクとして作成されています。

```
//レーダー処理の作成
prader_task = TaskMake(exec03_01_rader, 0x8000);
```

この時、表示するキャラクターを登録するために、このレーダー処理タスクのポインタを保持しておきます。

そして、ボールの処理作成時にボールを表示キャラクターとして登録します。

登録は、ボール処理タスクのポインタを配列に格納する形で行なわれます。

```
//レーダー処理のために登録
prader_task->Work[ loop ] = pnew_work;
```

登録が終了したら、次は実際のレーダー処理、座標の変換と表示の部分です。

処理は exec03_01_rader で一括して行なっています。

LIST 3 －15－1 exec03_01_rader

```
void exec03_01_rader(TCB* thisTCB)
{
#define RADER_POS_X  0
#define RADER_POS_Y  0

    EX3_1_STRUCT* pEX3_1pos_data;
    SPRITE2 sprt;
    int loop;
    RECT pix_rect = { 0, 0, 2, 2};

    sprt.X = RADER_POS_X;
    sprt.Y = RADER_POS_Y;
    sprt.SrcRect = NULL;
    SpriteDraw2(&sprt,2);

    sprt.SrcRect = &pix_rect;

    for( loop = 0; loop < EX3_1_MOVE_COUNT; loop++)
    {
        //ワークに格納されているのは表示する相手のデータポインタ
        pEX3_1pos_data =  (EX3_1_STRUCT*)thisTCB->Work[ loop ];

        //表示座標を取得
        sprt.X = pEX3_1pos_data->sprt.X;
        sprt.Y = pEX3_1pos_data->sprt.Y;
        //座標の変換を行う
        sprt.X = sprt.X / 10 + RADER_POS_X;
        sprt.Y = sprt.Y / 10 + RADER_POS_Y;
        //レーダーキャラをスプライトで表示
        SpriteDraw(&sprt,1);
    }
}
```

レーダー画面の表示

まず最初に、下地となる、レーダー画面そのものを表示します。

レーダー画面の大きさは64×48ピクセル、ちょうど画面の1／10となっています。表示方法も特殊な処理等はなく、単純な表示となります。

表示するキャラクターの座標を取得

次に、登録したキャラクターの座標を取得します。

これは、初期化時に格納したポインタから得られます。その後、得られた座標をレーダー画面に合わせて座標変換を行ないます。

レーダー画面は、先ほど書いた通り1／10ですので、変換計算もそれに合わせて座標を1／10にしてやります。

レーダー上にキャラを表示

最後にレーダー上での表示ですが、通常は小さな点か矩形を用います。

ここで、点や矩形を描画するAPIを使用しても良いのですが、点が小さすぎて見えにくいのと、点や矩形移外でも表示できる汎用性もあるため、ここではスプライトで表示を行なっています。

スプライトですので、処理もシンプルで、座標を設定して、表示関数を呼び出してやるだけです。

後は、登録されたキャラクターの数だけ処理を行なえば、レーダー処理は完成です。

3-16 現在のFPSを知る

FPSとは？

FPS とは、FramePerSecond の略で、1 秒間に何回画面を更新したかを示す言葉です。フレームレートと呼ばれることもあります。

ここでは、この FPS を計測方法を紹介します。

手法そのものは簡単で、1 秒間という時間を計るタイマーを用意し、その計測期間中に、何回画面を書き換えたかをカウントしてやります。

FPS表示プログラム

次に、プログラムの解説です。

まず初期化時に、Windows の API SetTimer を使用し、1 秒間に 1 度呼び出されるタイマーを作成しています。

タイマー内部では、1 秒ごとに更新された回数を内部の static 変数 record_time に記録しています。

g_Count はフレームの合計数ですので※、現在の数値から 1 秒前の数値を引いてやれば、1 秒間の書き換え回数が計算できます。

そして、表示用にグローバル変数 g_EX03_16_FramePerSecond に計算した数を書き込みます。

コールバックとは

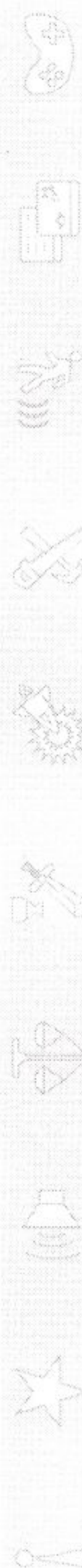
見てもらえれば分かるように、さほど難しい事はないのですが、コールバックという概念が必要になるため、少し詳しく解説します。

コールバックとは、特定の条件がそろった時に、外部から呼び出される処理の事です。

ここでは、EX03_16_time_callback 関数がそれに当たり、一定時間毎に呼び出されます。

設定は、SetTimer の第 4 引数で行なわれ、第 3 引数に、呼び出される時間の間隔をミリ秒単位で指定します。

サンプルでは、単純に 1 秒間の書き換え枚数ですが、精度を上げるために 10 秒毎の平均値を取る等、改良してみても良いでしょう。

※ g_Count は、システム側で用意された、今までに更新されたフレームの合計数。1 フレームに付き 1 ずつカウントされる。

LIST 3 −16−1 現在のFPSを知る

```
#define SECOND 1000.0
static int g_EX03_16_FramePerSecond;

void CALLBACK EX03_16_time_callback( HWND hwnd, UINT uMsg, UINT_PTR idEvent, 
DWORD dwTime)
{
static float record_time;
    //1秒間に更新されたフレーム数を記録
    g_EX03_16_FramePerSecond = g_Count - record_time;
    record_time = g_Count;
}

void init03_16(TCB* thisTCB)
{
    //1秒毎に呼び出す関数を設定する
    SetTimer( NULL, NULL, SECOND, EX03_16_time_callback );

    g_EX03_16_FramePerSecond = 0;
}

void exec03_16(TCB* thisTCB)
{
    float fps;
    char str[128];
    RECT font_pos = {  0,  0,640,480,};

    //FPSの表示
    sprintf( str,"現在のFPS %d",g_EX03_16_FramePerSecond);
    g_pFont->DrawText( NULL, str, -1, &font_pos, DT_LEFT, 0xffffffff);
}
```

3-17 文字を管理する

ゲーム中に表示するテキストの管理

通常、文字を表示する時は、プログラムの文字列として扱い、直接プログラム上に記述するでしょう。シューティングやアクション等の、文字表示が余り多く無いゲームでしたらこの方法で問題はありません。

しかし規模が大きくなったり、ストーリー進行に重要な意味のあるゲームでは、その文字表示量は増え、直接プログラムに記述すると、開発する際に少々効率が悪くなってきます。

そこで普通は、テキスト部分をファイル化して、データにまとめ、状況に応じて表示できるようにします。また、この手法ですと、文章データを作成する人とプログラマーの作業を分離する事ができ、作業の負荷を減らすことが出来ます。

#include で文字表示の処理をする

この時、変換処理のための、コンバータを作成しても良いのですが、その前にC言語のマクロ機能、#include を使う事を検討してみてください。

複雑な文字表示で無い限り、十分に対応できるかと思います。

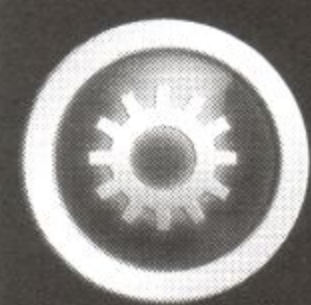
ただし、この方法で出来るのは、あくまで分岐などの無い、シンプルな文字表示までです※。

LIST 3 －17－1

```
char* MESS_DATA1[] =
{
#include "SCENE_A.TXT"
};

char* MESS_DATA2[] =
{
#include "SCENE_B.TXT"
};
```

テキストデータ例 SCENE_A.TXT

```
"サンプル文字列です",
"テキストファイル内では",
"このように文字列を記述していきます",
NULL,
```

※複雑な表意やシナリオの分岐などは、[3－31]で紹介するような、簡易言語を作成する必要があります。

3-18 得点の管理

ゲームにおける得点

シューティングやアクションでの得点（スコア）について考えてみます。

敵を倒すとそれに応じた、得点（スコア）が入るというゲーム性は、ユーザーにとって行動に対する結果が非常に分かりやすいシステムです。

スコアという形ではなくても、例えばRPG等では経験値という概念で同様のシステムを持っています。

しかし、一般的に、ゲームには1種類の敵だけが出てくる事はなく、何種類もの敵が出てきます。

また、スコアの種類も敵を倒すだけではなく、ボスを倒した時や時間によるタイムボーナス等、幾つも種類があります。

プログラムでの得点の管理方法

では、それらはどのように管理すればよいのでしょうか。

ここでは、IDによるスコア管理を考えてみます。

基本的な考えは単純で、スコアの種類を何種類かに分け、それぞれにIDを振って管理するというものです。

この手法だと、単純なスコア加算や時間によるボーナス等、シンプルなデータの計算であれば統一的な手順で管理できます。

得点管理のプログラム

では実際に、プログラムを見ていきます。

メインの処理自体はシンプルで、Xキーで IDを切り替え、Zキーで関数EX03_18_get_scoreにIDを渡し、返り値をスコアとして加算します。

その後、加算したスコアやIDを画面に表示しています。

次に、実際の処理関数EX03_18_get_scoreです。

まず、取得したIDを元に、スコアテーブルから加算するスコアを取得します。

この際、取得する値に応じて、単純な加算ではなく特殊な処理を行なうようにしています。

プログラムでは、switch文で、その処理を行なっています。

もし、特殊なスコア加算を行なう場合はこの部分を改良すると良いでしょう。

最後に、取得したスコアを返して、処理は終了です。

なお、ここでは処理の理解を優先しているためIDのみを渡していますが、IDだけで処理が終了

するケースはあまり多くありません。

そのため、実際に使用する時は、ID以外にもスコア演算用のデータを渡すようにすると良いでしょう。

LIST 3 ―18―1 得点の管理

```
typedef struct{
    int             Score;
    int             ScoreID;
} EX03_18_STRUCT;

int EX03_18_get_score( int score_id )
{
#define SPECIAL_SCORE1 -1
#define SPECIAL_SCORE2 -2
static int ScoreTable[] =
{
    100,
    1000,
    SPECIAL_SCORE1,
    2550,
    1280,
    500,
    SPECIAL_SCORE2,
    10,
};
    int score;


    score = ScoreTable[ score_id ];

    if( score < 0 )
    {//負数なら特殊スコアとしてスコア計算
        switch ( score )
        {
            case SPECIAL_SCORE1: score = g_Count * 10; break;
            case SPECIAL_SCORE2: score = 10000 + g_Count * 50; break;
        }
    }
```

```
    return score;
}

void exec03_18(TCB* thisTCB)
{
    EX03_18_STRUCT* work = (EX03_18_STRUCT*)thisTCB->Work;
    RECT font_pos = {  0,  0,640,480,};
    char str[128];

    //Zキーでスコアを加算
    if( g_DownInputBuff & KEY_Z ) work->Score += EX03_18_get_score( work-
>ScoreID );
    //XキーでIDを切り返る
    if( g_DownInputBuff & KEY_X )
    {
        work->ScoreID++;
        work->ScoreID &= 0x07;
    }

    //スコアの表示
    sprintf( str,"SCORE: %8d  ID%d¥n加算スコア: %8d",
        work->Score,work->ScoreID,EX03_18_get_score( work->ScoreID ));
    g_pFont->DrawText( NULL, str, -1, &font_pos, DT_LEFT, 0xffffffff);
}
```

その他のスコア管理方法

最後に、スコア管理はここに挙げた方法だけで全てOKという訳ではありません。

もし、敵を倒すだけでスコアが加算されるといったシンプルなシステムならば、こういった管理システムはかえって面倒な事になってしまいます。

また、ゲームによっては複雑な得点システムを用いたり、IDだけでは対処しきれない独自のボーナスシステムを作成する場合もあります。

この場合は、各ゲーム個別に対処するしかないため、専用のスコア処理を作成したほうが良いでしょう。

3-19 単純なセーブ・ロード

セーブとロードの処理方法

セーブ・ロードの使いどころ

セーブ・ロード処理について考えてみます。

セーブ・ロード処理はほぼ全てのゲームが行なう処理と言って良いでしょう。

RPGやアドベンチャーではゲームの進行状態記録するのになくてはならない機能ですし、一見必要のなさそうなクションやシューティング等でも、ゲームの設定等を記録しておく時などに使います。

また上記のような、一般的な使い方以外でも、出現した隠しキャラや隠しステージの記録、デバッグ時の状態記録等、様々に使用されます。

上記のように、様々な箇所で使われますが、仕組み自体は単純で、Cの標準関数であるfreadとfwriteを呼び出すだけです。

しかし、ゲームジャンルや内容によって、その記録の手順が変わってしまうので、少々面倒な処理でもあります。

そこでここでは、いくつかに分けて、セーブ・ロードの処理と概念を紹介していきます。

スコアデータの記録・読み込み

まずは単純なセーブ・ロードの例として、シューティング等のスコアデータの記録・読み込み処理を作成してみましょう。

処理の流れは非常にシンプルで、配列に定義してあるスコアデータをそのまま、記録&読み込みしてやります。

ただ、このままではシンプルすぎるので、起動時にデータの有無を調べて、自動でセーブデータを読み込む「オートロード」処理も加えてあります。

では、早速プログラムを見ていきます。

初期化

まずは初期化です。最初にオートロードとして、すでにセーブされたデータが無いか読み込みに行きます。

もしデータが無ければ、エラーが帰ってきますので、まだセーブはされていないと判断し、初期化用のデータを読み込みます。

メイン処理

次にメイン処理です。サンプルではZキーでロード、Xキーでセーブを行ないます。

方向キーで記録するスコアの変更が可能ですので、色々変更して、セーブ・ロードしてみてください。

セーブ・ロード関数

最後に、肝心のセーブ・ロード関数です。

セーブ関数 EX03_19_score_save は指定したスコア記録データを指定のファイル名で書き込みます。

スコアデータは1～5位までを記録しているため、書き込むデータ数も5つになります。

同様に、ロード関数 EX03_19_score_load は格納データへのポインタに指定したスコア記録ファイルを読み込みます。

両関数とも、非常にシンプルですので理解が難しい所は無いでしょう。

LIST 3 －19－1 スコアデータのセーブ・ロード

```
#define RANK_COUNT 5
#define DATA_FILE_NAME "EX03_19_SCORE_DATA.DAT"

typedef struct{
    int             Score;
    char            Name[4];
} EX03_19_STRUCT;

static EX03_19_STRUCT  gEX03_19_SAVE_DATA[ RANK_COUNT ];


void EX03_19_score_save( char* FILE_NAME, EX03_19_STRUCT* SaveData )
{
    FILE* save_file;

```

```
    //バイナリ書き込みモードでファイルをオープン
    save_file = fopen( FILE_NAME, "w+b");

    //スコアデータをセーブする
    fwrite( SaveData, sizeof( EX03_19_STRUCT ) , RANK_COUNT, save_file );

    //ファイルを閉じる
    fclose( save_file );
}

int EX03_19_score_load( char* FileName, EX03_19_STRUCT* LoadData )
{
    FILE* load_file;

    //バイナリ読み込みモードでファイルをオープン
    load_file = fopen( FileName, "r+b");

    //もしファイルがなければエラーを返す
    if( load_file == NULL )return true;

    //スコアデータをロードする
    fread( LoadData, sizeof( EX03_19_STRUCT ) ,RANK_COUNT, load_file );

    //ファイルを閉じる
    fclose( load_file );

    //正常終了
    return false;
}

void init03_19(TCB* thisTCB)
{
    //初期化時にスコアをロードを試みて、ファイルがなければデフォルトのデータを読み込む
    if( EX03_19_score_load( DATA_FILE_NAME, gEX03_19_SAVE_DATA ) )
    {//デフォルトのスコアをロードする
        EX03_19_score_load( "EX03_19_DEFAULT_DATA.DAT", gEX03_19_SAVE_DATA
);
    }
}
```

```
void exec03_19(TCB* thisTCB)
{
    int loop;
    RECT font_pos = {   0,   0,640,480,};
    char str[128];

    //Zキーでスコアをロード
    if( g_DownInputBuff & KEY_Z )
        EX03_19_score_load( DATA_FILE_NAME, gEX03_19_SAVE_DATA );
    //Xキーでスコアをセーブ
    if( g_DownInputBuff & KEY_X )
        EX03_19_score_save( DATA_FILE_NAME, gEX03_19_SAVE_DATA );


    //方向キーでスコアの増減を行なう
    if( g_DownInputBuff & KEY_UP)
    {
        for( loop = 0; loop < RANK_COUNT; loop++ )
        {
            gEX03_19_SAVE_DATA[ loop ].Score += 100;
        }
    }
    if( g_DownInputBuff & KEY_DOWN)
    {
        for( loop = 0; loop < RANK_COUNT; loop++ )
        {
            gEX03_19_SAVE_DATA[ loop ].Score -= 100;
        }
    }

    //スコアデータの表示
    for( loop = 0; loop < RANK_COUNT; loop++ )
    {
        sprintf( str,"RANK%d: %8d  %s",
            loop+1, gEX03_19_SAVE_DATA[ loop ].Score,gEX03_19_SAVE_DATA[
loop ].Name);
        g_pFont->DrawText( NULL, str, -1, &font_pos, DT_LEFT, 0xffffffff);
        font_pos.top += 32;
    }
```

```
        font_pos.top += 32;
        g_pFont->DrawText( NULL,
            "Zキーでロード、Xキーでセーブ¥n方向キーでスコア変更",
            -1,
            &font_pos,
            DT_LEFT,
            0xffffffff);
}
```

3-20 少し複雑なセーブ・ロード

様々な設定や状況をセーブ・ロードする

続いて、もう少し複雑なセーブ・ロード処理について考えてみます。

[3－19]では、スコアデータだけでしたので、指定の構造体を読み書きするだけでしたが、実際の処理ではそれだけではすまない事があります。

例えば、ゲーム設定をセーブした場合を考えてみましょう。

その設定が単なるフラグだけでしたら問題は無いのですが、画面の色や背景の変更等、何らかの処理が必要な場合があります。

また、スコアデータのように単純なデータ構造が１つであれば、セーブ・ロード処理はさほど難しくないのですが、RPGやアドベンチャーゲームでは、ゲームの進行状況や多数のフラグの状態を記録し、ロード時にはその状態を反映しなくてはなりません。

実際、各処理毎に構造体を割り振って、セーブロード処理をバラバラに行なうと、いささか面倒な処理になってしまいます。

どのように処理すればいいか

この問題の解決方法として、セーブデータを１つの構造体としてまとめてしまうという手法があります。

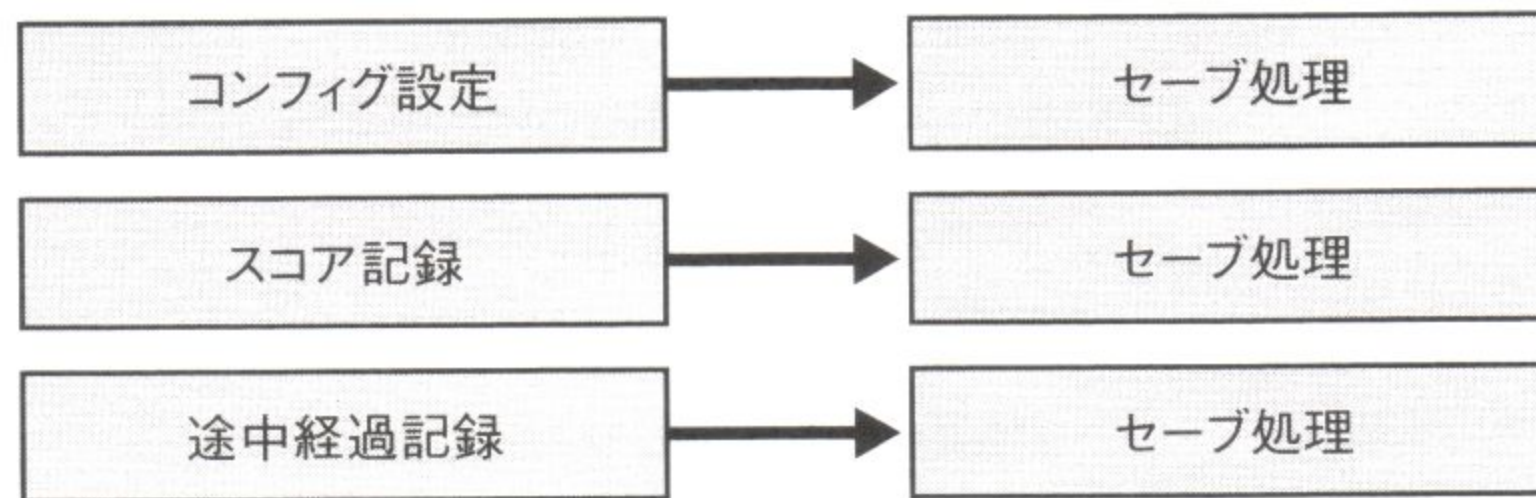
処理別にデータ構造体を用意するのではなく、セーブデータという１つの構造体に必要なデータをまとめておくのです。

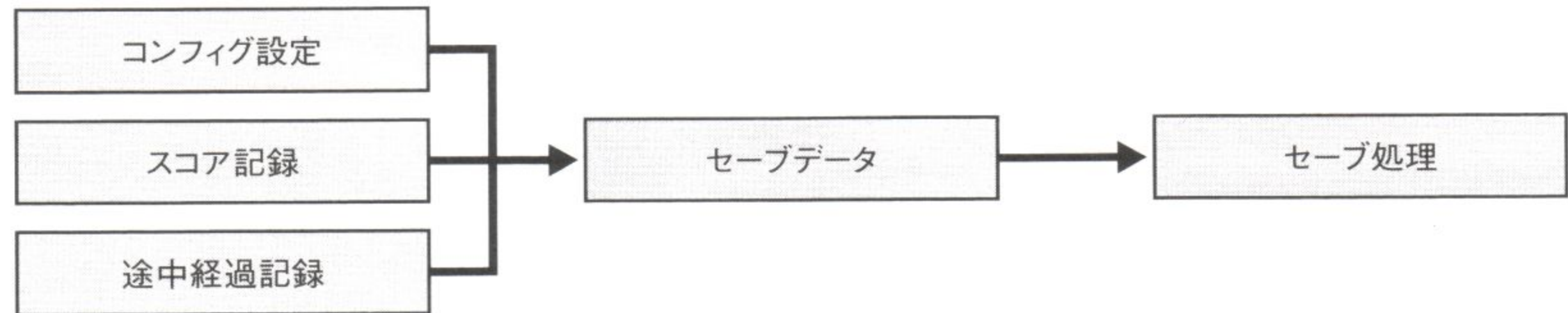
データをセーブする必要がある処理は、この構造体と間接的にやり取りを行ない、自分でデータのセーブは行ないません。

ただし、セーブが必要な処理は必ず、このデータとやり取りを行なわなくてはならないため、若干処理手順は増えてしまいます。

こうしておけば、セーブに関する処理を１箇所にまとめる事ができ、[3－19]のような単純なセーブ・ロードと、あまり変わらなくなります。

図3 －20－1 データをまとめておき、そこに各処理が間接的にアクセスするイメージ図

コンフィグ設定	→	セーブ処理
スコア記録	→	セーブ処理
途中経過記録	→	セーブ処理

各処理が個別にセーブすると複雑化し、効率が悪い

コンフィグ設定				
スコア記録	→	セーブデータ	→	セーブ処理
途中経過記録				

セーブデータとして1箇所にまとめることで単純化を図る

プログラムの解説

データを構造体にまとめる

では実際のプログラムを見ていきましょう。

まずは、セーブ用に構造体をまとめます。

ここでは、[3－19]で使用したスコア記録用の構造体と、ゲームの設定用の変数として自機数と難易度を１つのセーブ用構造体にまとめています。

初期化とメイン処理

次に初期化ですが、ここではオートロード等の処理は行なわず、初期化用のデータを読み込むだけです。

その後、メインの処理では、Ｚキーでセーブ、Ｘキーでロードを行ないます。

ロード処理

ロード処理ですが、まず、指定した構造体にロードを行ないます。

その後、ロードしたデータをゲーム上に反映するために、関数 EX03_20_load_update を呼び出しています。

本来、関数内では設定の反映処理を行なうのですが、内容が広がりすぎてしまうため、解説は省略します。

その他、関数内では簡単なエラーチェックを行なっています。

これは、ロードしたデータが破損していたり、改造されていたりした場合の対応策です。

もし、データが破損していた場合は、初期化用のデータを読み込んで、バグの発生を防いでいます。

セーブ処理

次にセーブの処理です。

こちらは、ロード時と違い、セーブ処理の直前に更新関数EX03_20_save_updateを呼び出しています。

この関数で、これからセーブするデータ構造体へ、データを更新します。

こちらは、エラー処理などは無く、更新後はセーブ処理の関数を呼び出して終了です。

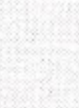
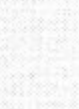
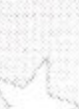
なお、セーブ・ロードの各関数ですが、記録する構造体以外は殆ど変わっていません。

内部では、まとめた記録用のデータ構造体を1つだけ、書き込み・読み込みする様になっています。

[3－19]同様、この部分で特に難しい所は無いと思います。

LIST 3－20－1 少し複雑なセーブ・ロード

```
#define RANK_COUNT 5
#define DATA_FILE_NAME "EX03_20_SAVE_DATA.DAT"

typedef struct{
    EX03_19_STRUCT ScoreData[ RANK_COUNT ];
    char           MyShipCount;
    char           Difficulty;
} EX03_20_STRUCT;

static EX03_20_STRUCT  gEX03_20_SAVE_DATA;


void EX03_20_save( char* FILE_NAME, EX03_20_STRUCT* SaveData )
{
    FILE* save_file;

    //バイナリ書き込みモードでファイルをオープン
    save_file = fopen( FILE_NAME, "w+b");

    //データをセーブする
```

```
    fwrite( SaveData, sizeof( EX03_20_STRUCT ) , 1, save_file );

    //ファイルを閉じる
    fclose( save_file );
}

void EX03_20_load( char* FILE_NAME, EX03_20_STRUCT* LoadData )
{
    FILE* load_file;

    //バイナリ読み込みモードでファイルをオープン
    load_file = fopen( FILE_NAME, "r+b");

    //データをロードする
    fread( LoadData, sizeof( EX03_20_STRUCT ) , 1, load_file );

    //ファイルを閉じる
    fclose( load_file );
}

int EX03_20_load_update( void )
{
    int loop;
    //ロード後にデータをプログラムに反映させる処理の例
//  SetMyShipCount( gEX03_20_SAVE_DATA.MyShip );
//  SetDifficulty ( gEX03_20_SAVE_DATA.Difficulty );

    //スコアの整列を調べる（簡単なチェック）
    for ( loop = 0; loop < RANK_COUNT - 1; loop++ )
    {
        if( gEX03_20_SAVE_DATA.ScoreData[ loop ].Score <
            gEX03_20_SAVE_DATA.ScoreData[ loop+1 ].Score )
        {
            return true;
        }
    }
    return false;
}
void EX03_20_save_update( void )
{
```

```
        //セーブ前にデータを更新しておく処理
}

void init03_20(TCB* thisTCB)
{
        //デフォルトのデータをロードする
        EX03_20_load( "EX03_20_DEFAULT_DATA.DAT", &gEX03_20_SAVE_DATA );

}

void exec03_20(TCB* thisTCB)
{
        int loop;
        RECT font_pos = {   0,   0,640,480,};
        char str[128];

        //Zキーでロード
        if( g_DownInputBuff & KEY_Z )
        {//ロード
                EX03_20_load( DATA_FILE_NAME, &gEX03_20_SAVE_DATA );
         //ロード後に、処理を行ない、エラーならデフォルトデータを読み込む
                if( EX03_20_load_update() )
                    EX03_20_load( "EX03_20_DEFAULT_DATA.DAT", &gEX03_20_SAVE_DATA );
        }

        //Xキーでセーブ
        if( g_DownInputBuff & KEY_X )
        {//セーブ前に、データの更新処理
                EX03_20_save_update();
         //セーブ
                EX03_20_save( DATA_FILE_NAME, &gEX03_20_SAVE_DATA );
        }


        //スコアデータの表示
        for( loop = 0; loop < RANK_COUNT; loop++ )
        {
                sprintf( str,"RANK%d: %8d  %s",
                                loop+1,
                                gEX03_20_SAVE_DATA.ScoreData[ loop ].Score,
```

```
                gEX03_20_SAVE_DATA.ScoreData[ loop ].Name );

        g_pFont->DrawText( NULL, str, -1, &font_pos, DT_LEFT, 0xffffffff);
        font_pos.top += 32;
    }

        font_pos.top += 32;
        g_pFont->DrawText( NULL,
            "Zキーでロード、Xキーでセーブ",
            -1,
            &font_pos,
            DT_LEFT,
            0xffffffff);
}
```

3-21 より柔軟なセーブ・ロードのシステム

いつでも、どこでもセーブしたい場合

最後に、もっと自由なセーブ・ロード処理について考えてみましょう。

今まで解説した、セーブ処理は、セーブ・ロードをする場所が固定的、例えばゲームの開始時や、セーブポイント等がある事を前程としての解説でした。

しかし実際のゲームでは、もっと自由なセーブをする事があります。

RPG などでは何処でもセーブできる機能をもった物がありますし、アドベンチャーやシミュレーションでは、現在の状況を簡単に保存するクィックセーブ機能等もその１つです。

これらのセーブ・ロードは、実際に記録しなくてはならない情報以外にも、現在のゲーム上での情報を記録する必要があります。

例えば、RPG ではプレイヤーの現在の表示座標等がそうですし、もし画面上を動く敵や町の人などがいれば、それらのステータスや座標も記録しなくてはなりません。

もちろん、こういった細かい記録を行なわなくても良いのですが、その場合ロード・セーブを繰り返した攻略や、ロード時の画面の違和感などにつながります。

そのため、こういった自由なセーブ・ロードをするのであれば、できるかぎり再現した方が望ましいでしょう。

ただ、想像がつくと思いますが、こういった処理は細かい座標やステータスの管理が面倒なため、いささか厄介です。

どのような処理を考えればいいか

こういった処理の、対処方法は大きく分けて２つあります。

１つは、すでに説明した、「１つの構造体で全てのステータスを管理する」という手法です。

これは、セーブに必要なデータ以外でも、ゲーム上で必要なデータや変数を全てを、１箇所にまとめて処理を行なうという事です。

こうすると、セーブ・ロード自体はシンプルで済みますが、ゲーム上で処理する内部的な変数でも保存しなくてはならない場合があるため、面倒さは増えます。

もう１つは、各処理が必要なセーブ・ロード処理を行なう機構を設けて、セーブ・ロードのタイミ

ングで、それらの処理を行なうものです。

この手法を使う時は、タイミングを管理する処理が必要になります。

図3 －21－1 手法1 一箇所にデータをまとめておき、そこに各処理が間接的にアクセスするイメージ図(3－20－1)

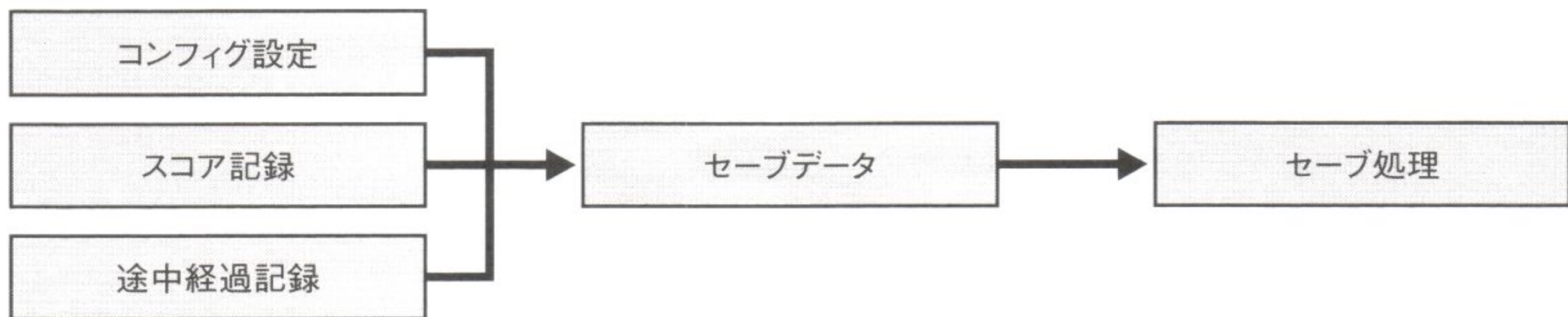
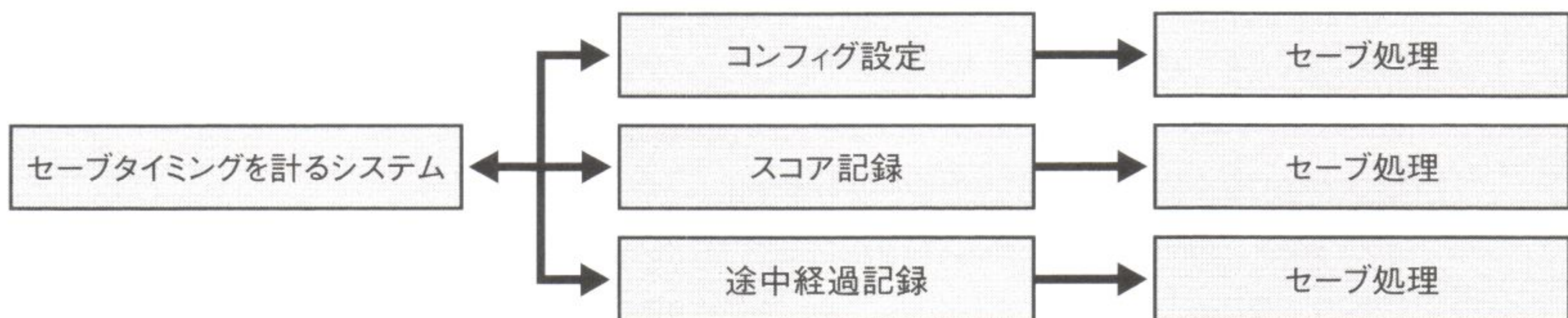



図3 －21－2 手法2 それぞれの処理が、細かいセーブ・ロード処理機能を有しており、タイミングで処理を行なうイメージ図

それぞれ一長一短があり、どちらが良いのかはゲームによりますので、一概には言えません。

また、双方の特徴を合わせたハイブリット型の手法を使用する場合もあります。

こういった実行環境やデータの記録・読み込みを行なう処理はそれぞれ「シリアライズ」「デシリアライズ」と呼ばれています。

もし興味があれば調べてみるとよいでしょう。

3-22　データを暗号化する

暗号化の必要性

ここではデータを暗号化する方法を紹介しましょう。

その前に、どういう状況でゲームに暗号化が必要なのかを考えてみます。

そもそも、暗号化というものは、その内容を相手に知られたくない時に用いられます。

ゲームにおいて知られたくないというものは、主にゲーム内容に関するものでしょう。

例えば、アドベンチャーゲームのテキストデータ等がそうですが、これをを暗号化しておき、ネタばれを防ぐといった用途が考えられます。

また、セーブデータやグラフィックデータの内容が直接見えてしまうと、改造されたり、意図しない用途で使用されてしまう可能性もあります。

そのため、データに対して暗号化を行なう事で、そういった事を防ぐ事が出来ます。

どのように暗号化するか

さて、暗号化の手順ですが、大きく2つの段階に分けられます。

すなわち、元のデータを暗号化する処理と、暗号化したデータを元に戻す処理（復号化）です。

少し難しく書きましたが、要するに、元のデータに何らかの計算式を用いて加工を行ない（暗号化）、その計算式と逆の結果をもたらす計算式を用いて元に戻す（復号化）という事です。

この際用いる計算式に、特定の数値を使用すると、この数値がいわゆる「暗号キー」になります。

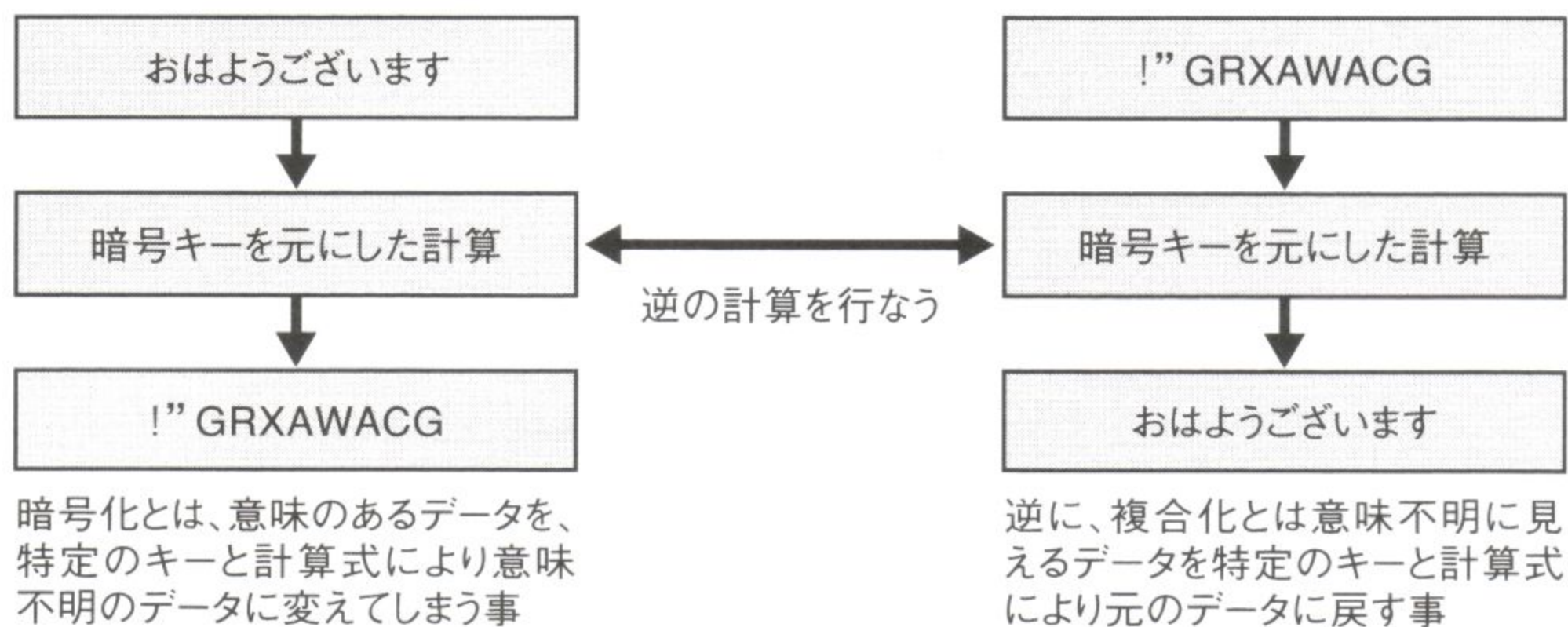
図3－22－1 暗号化と復号化の関係のイメージ図

おはようございます
↓
暗号キーを元にした計算
↓
!" GRXAWACG

暗号化とは、意味のあるデータを、特定のキーと計算式により意味不明のデータに変えてしまう事

（↔ 逆の計算を行なう）

!" GRXAWACG
↓
暗号キーを元にした計算
↓
おはようございます

逆に、複合化とは意味不明に見えるデータを特定のキーと計算式により元のデータに戻す事

暗号化のプログラム

では実際のプログラムを見ていきましょう。

サンプルでは元のデータを、２種類の方法で暗号化して画面に表示しています。

加算、減算による暗号化

まず、１つ目の暗号化ですが、これは計算に「加算」を用いています。

これは元のデータに、暗号化キーを加算するだけの非常に単純なものです。

そして複合化は、これとは逆の計算すなわち「減算」を使います。

こちらも非常に単純で、暗号化されたデータから、暗号化キーを減算するだけです。

以上の処理で、データの暗号化、複合化は終了です。

もちろんこれほど単純ですと、データを暗号化しても、簡単に見破られてしまうのですが、暗号化と複合化の手順は理解しやすいかと思います。

乱数を用いた暗号化

さて２つ目は、少し複雑な暗号化を行なってみます。これは計算式に、乱数を用いたものです。

乱数を生成する関数 rand は、乱数の初期化関数 srand に数値を渡す事で、得られる数値の順序を同じにする事が出来ます。

この性質を使用して、srand に渡す数値を暗号キーとし、得られる数値を計算に利用します。

計算式そのものは、論理演算の「排他的論理和」を用いて暗号化、複合化を行なっています。

この論理演算は、同じ数値どうしで２回演算を行なうと、元の数値に戻るという性質を持っています。

そのため、暗号化と複合化で、同じ式が使えるのが特徴です。

LIST 3 －22－1 生データを暗号化する

```
void exec03_22(TCB* thisTCB)
{
#define NUM_DATA_COUNT 4
#define CODE_KEY 0xAA
#define RAND_CODE_KEY 0x10
	//元データ
	unsigned char  in_num_data[]  = {0x00,0x01,0x02,0x03,};
	//暗号化データ格納用変数
	unsigned char  encode_num_data1[ NUM_DATA_COUNT ];
```

```
unsigned char  encode_num_data2[ NUM_DATA_COUNT ];
//復号化データ格納用変数
unsigned char  decode_num_data1[ NUM_DATA_COUNT ];
unsigned char  decode_num_data2[ NUM_DATA_COUNT ];

char  str[128];
unsigned char  code_key;
int  loop;
RECT font_pos = {  0,  0,640,480,};


//暗号化処理1
for( loop = 0; loop < NUM_DATA_COUNT; loop++)
{//暗号化の演算を行なう
    encode_num_data1[ loop ] = in_num_data[ loop ] + CODE_KEY;
}
//復号化処理1
for( loop = 0; loop < NUM_DATA_COUNT; loop++)
{//暗号化とは逆の演算を行なう
    decode_num_data1[ loop ] = encode_num_data1[ loop ] - CODE_KEY;
}

//暗号化処理2
srand( RAND_CODE_KEY );
for( loop = 0; loop < NUM_DATA_COUNT; loop++)
{//暗号化の演算を行なう
    encode_num_data2[ loop ] = in_num_data[ loop ] ^ rand() % 255;
}
//復号化処理2
srand( RAND_CODE_KEY );
for( loop = 0; loop < NUM_DATA_COUNT; loop++)
{//暗号化とは逆の演算を行なう
    decode_num_data2[ loop ] = encode_num_data2[ loop ] ^ rand() % 255;
}


//元データ表示
sprintf( str,"暗号前データ  :0x%02x 0x%02x 0x%02x 0x%02x",
in_num_data[0], in_num_data[1], in_num_data[2], in_num_data[3]);
g_pFont->DrawText( NULL, str, -1, &font_pos, DT_LEFT, 0xffffffff);
```

```
    font_pos.top += 64;

    //暗号データ1表示
    sprintf( str,"暗号化データ1：0x%02x 0x%02x 0x%02x 0x%02x",
    encode_num_data1[0], encode_num_data1[1],
    encode_num_data1[2], encode_num_data1[3]);
    g_pFont->DrawText( NULL, str, -1, &font_pos, DT_LEFT, 0xffffffff);
    font_pos.top += 32;

    //複合データ1表示
    sprintf( str,"復号化データ1：0x%02x 0x%02x 0x%02x 0x%02x",
    decode_num_data1[0], decode_num_data1[1],
    decode_num_data1[2], decode_num_data1[3]);
    g_pFont->DrawText( NULL, str, -1, &font_pos, DT_LEFT, 0xffffffff);
    font_pos.top += 64;


    //暗号データ2表示
    sprintf( str,"暗号化データ2：0x%02x 0x%02x 0x%02x 0x%02x",
    encode_num_data2[0], encode_num_data2[1],
    encode_num_data2[2], encode_num_data2[3]);
    g_pFont->DrawText( NULL, str, -1, &font_pos, DT_LEFT, 0xffffffff);
    font_pos.top += 32;

    //複合データ2表示
    sprintf( str,"復号化データ2：0x%02x 0x%02x 0x%02x 0x%02x",
    decode_num_data2[0], decode_num_data2[1],
    decode_num_data2[2], decode_num_data2[3]);
    g_pFont->DrawText( NULL, str, -1, &font_pos, DT_LEFT, 0xffffffff);


}
```

3-23 簡易スクリプトを作る－シューティング編 1

スクリプトを使うメリット

シューティングゲームにおいて、敵の動きを直接プログラムするのは、なかなか大変です。

単純な動きは良いのですが、複雑な動きをプログラムした場合は、動きの修正自体も大変になってきます。

そこで、通常こういった動きをたくさん作成する場合は、動きを制御するスクリプト(簡易言語)を作成します。

スクリプト化する事の代表的なメリットとして、以下のようなものが挙げられます。

1・敵の動きをスクリプトという形で、データ化できる

2・動きの修正が容易になる

3・スクリプトの仕様を上手に作ればプログラマ以外でも動きのデータ作成が可能になる

どう実現するか

メリットが分かった所で、実際どのように処理すればよいのかを考えて見ましょう。

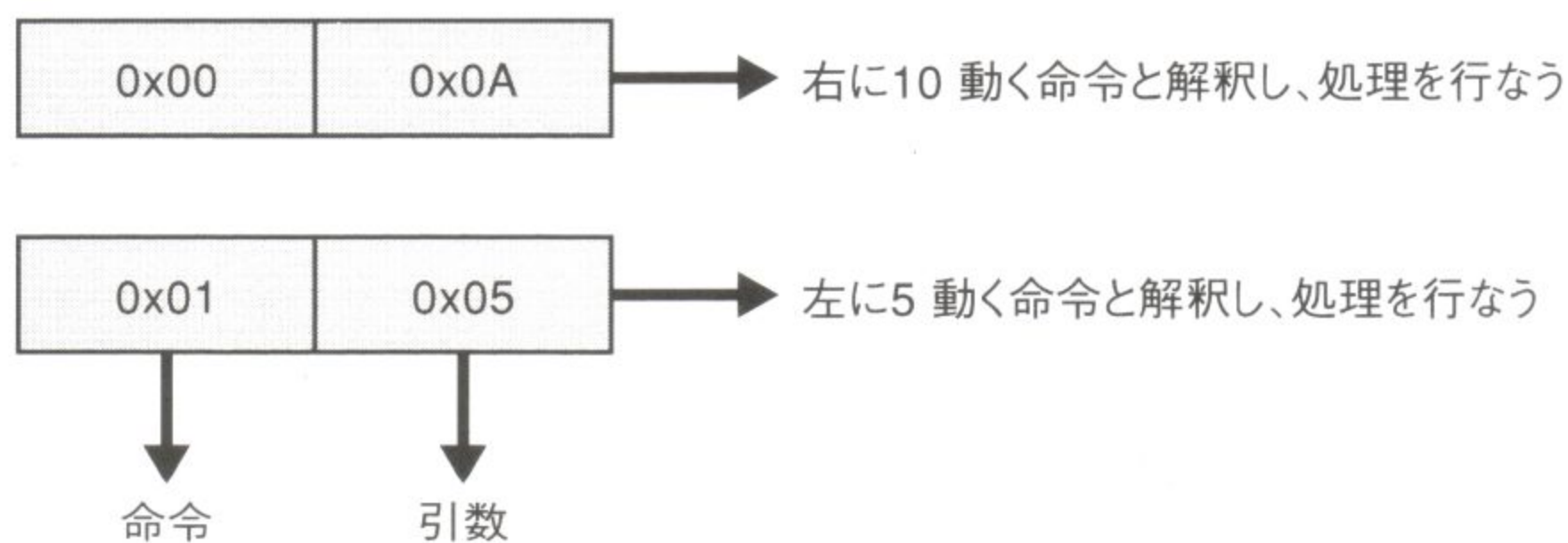
実は基本となる概念自体は難しくありません。

プログラマーが定義した数値データを「命令」とみなし、この命令に対応した処理を行ないます。

この時に、命令だけでは不便なため、引数となるデータも同時に渡してやるようにします。

情報処理をかじった人は、「敵の動きを解釈する仮想マシン」と考えると理解しやすいでしょうか。

図3－23 数値を命令とみなして、処理をさせる

0x00	0x0A	→ 右に10 動く命令と解釈し、処理を行なう
0x01	0x05	→ 左に5 動く命令と解釈し、処理を行なう
↓命令	↓引数	

3-24 簡易スクリプトを作るーシューティング編2

スクリプトの仕様を決める

まず、実際にプログラムを作成する前に、スクリプトの仕様を決めなくてはなりません。

具体的には、「どのような動きをさせたいか？」「その動きは引数でどのように変わるのか？」等を定義します。

この仕様作りが上手くないと、単純な動きの処理でもかえって面倒になったしまったり、データや動きの変更が難しくなってしまったりするので、よく吟味して決定しなくてはなりません。

この仕様に基づき、各処理の実装を行ないます。

サンプルプログラムの仕様

ここでは、処理の理解が目的ですので、行動命令はシンプルに、弾の発射と左右の移動のみを実装します。

左右への移動は、距離を調整できると便利ですので、引数から移動時間を取得するようにします。

また、それ以外にも必要と思われる命令として、動きを停止する命令、処理の終了状態を表す命令、スクリプトを最初から繰り返すリセット命令の３つを実装します。

まとめた命令の一覧は以下のようになります。

図3ー24ー1 命令の一覧表

命令		
0x00	CODE_END:	命令の終了状態を表し、次の命令を読み込む
0x01	CODE_STOP:	引数の時間だけ処理を停止
0x02	CODE_LEFT:	引数の時間だけ左に進む
0x03	CODE_RIGHT:	引数の時間だけ右に進む
0x04	CODE_SHOOT:	弾を発射する
0x05	CODE_RESET:	行動をリセットし、処理を初めから開始する

スクリプトの処理方法

次にこの数値をどうやって定義し、処理を行なうかです。

本来、こういった処理を行なうためにには、テキスト等で記述したスクリプトデータを、専用のコンパイラを作成して処理をしたりします。

ですが、今回の仕様では、そこまで本格的な処理が要求される訳でも無く、また手順としても少々大げさすぎます。

そこで、本書ではもっと手軽に、C 言語のマクロ機能で命令を定義し、それを配列上にデータとして並べて解釈する、という手法で処理を行ないます。

様式としてはいささか乱暴ですが、手軽に言語定義ができる手法としては悪くないと思います。

3-25 簡易スクリプトを作るーシューティング編３

スクリプトのプログラム

ではいよいよプログラムを見ていきましょう。

LIST 3 －25－1 簡易スクリプト

```
typedef struct{
	SPRITE			Sprt;
	int			PC;		//コードの実行位置を示すカウンタ
	unsigned char*	pCode;		//コードデータを格納してあるポインタ
	unsigned char	Code;		//現在実行中の命令
	unsigned char	Data;		//現在実行中の命令の引数
} EX03_SCRIPT_SHOOT;

#define CODE_END		0x00
#define CODE_STOP		0x01
#define CODE_LEFT		0x02
#define CODE_RIGHT		0x03
#define CODE_SHOOT		0x04
#define CODE_RESET		0x05

#define MOVE_SPEED		8


void init03_script(TCB* thisTCB)
{
	EX03_SCRIPT_SHOOT* work = (EX03_SCRIPT_SHOOT*)thisTCB->Work;

	//使用するテクスチャの読み込み
	D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥0044.png",&g_pTex[0] );
	D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥0055.png",&g_pTex[1] );

	//座標と、実行コードの初期化
	work->Sprt.X = SCREEN_WIDTH  / 2;
```

```
    work->Sprt.Y = SCREEN_HEIGHT / 3;

    work->Code = CODE_END;

}

void exec03_script(TCB* thisTCB)
{
    EX03_SCRIPT_SHOOT* work = (EX03_SCRIPT_SHOOT*)thisTCB->Work;

unsigned char  script_data[] =
{//行動イベントデータ
    CODE_LEFT  , 0x10,  //左移動
    CODE_RIGHT , 0x20,  //右移動
    CODE_STOP  , 0x10,  //停止
    CODE_SHOOT , 0x00,  //弾の発射
    CODE_LEFT  , 0x10,  //左移動
    CODE_RIGHT , 0x18,  //右移動
    CODE_SHOOT , 0x00,  //弾の発射
    CODE_LEFT  , 0x18,  //左移動
    CODE_RESET , 0x00,  //行動をリセットし、最初に戻る
};


    //実行するスクリプトデータ
    work->pCode  = script_data;

    //スクリプトの実行
    EX03_script_exec( work );

    //表示
    SpriteDraw( &work->Sprt , 0);
}
```

初期化

まず、サンプルを元に、スクリプトの使用例と全体の概略から見ていきます。

最初は初期化です。移動するキャラの座標と、スクリプト関数に最初に実行させる命令を登録します。

この命令は、スクリプトデータとは一切関係なく最初の１度だけ実行されます。

ここでは、実行を終了して次の命令(ここでは実際のスクリプトデータ)を読み込むCODE_END を指定しています。

メイン処理

次にメインですが、スクリプトの実行と、キャラの表示という非常にシンプルかつ短い物になっていまが、これは実際の行動は全て、スクリプトデータ側で定義されているからです。

スクリプトデータの構造

そのデータですが、[3－24]で解説したとおり、スクリプトのデータ構造はバイト単位の単純な配列になっており、構造は以下の通りになっています。

図3－25－1 スクリプトデータ配列の構造

偶数アドレス	奇数アドレス
命令	引数

シンプルな1 次元配列で、命令と引数が交互にならんでいる

命令と引数データが、常にペアで配置されるようになっており、このペアで1つの行動を示します。

引数の内容は命令によって変わりますが、基本的には指定した時間だけ命令を実行し続けます。

例えば、 CODE_LEFT, 0x10 と記述すると、左へ16 フレームの間移動し続け、CODE_STOP, 10 だと、10 プレームの間停止します。

これらの行動を、組み合わせて、キャラの行動を組み上げていきます。

より複雑な動きを表現するには

命令にあるのは単純な行動ばかりなので、複雑な動きや個別の動きを表現するのには限界があります。

その時はメイン側のプログラムに追加の行動処理を書き加えるか、スクリプトに新規に命令を追加する必要があります。

また、実行するスクリプトデータを、途中で切り替えても良いでしょう。

3-26 簡易スクリプトを作る－シューティング編4

簡易スクリプトの実装部分

最後に簡易スクリプトの実装部分を解説します。

スクリプトデータは、関数 EX03_script_exec で実行されます。

この関数は引数として、実行時の情報と処理内容を保存する構造体、EX03_SCRIPT_SHOOT を要求します。

この構造体は、実行される命令とデータを、常に内部の変数 Code と Data に格納しています。

さて、実際のプログラム部分ですが、これは、1フレームに1度だけ、実行中の命令を処理します。

命令の解釈は switch 文で行なわれ、処理の結果命令が終了したら、終了命令 CODE_END を内部で発行し、実行中の命令を終了します。

この命令は、次に処理する命令とデータを読み込む命令でもあり、あとはこれを処理が終了するまで繰り返します。

スクリプトの命令と処理内容

以下に各命令の簡単な処理内容を解説を示します。

CODE_END

行動の終了時に次の命令を読み込む命令です。

構造体内部の変数に、命令と、命令に対する引数データを読み込みます。

同時に次に実行する命令を示す様、PC を1つ進めます。

CODE_STOP

指定の時間、行動を停止する命令です。

時間として渡された引数データを毎フレーム減算していき、0になったら行動処理を終了します。

CODE_LEFT

指定の時間、左に移動します。

CODE_RIGHT

指定の時間、右に移動します。

移動方向の違いを除いては同じ命令です。時間として渡された引数データ分だけ、それぞれの方向に移動する様、表示位置を更新しています。

CODE_SHOOT

弾を発射します。

タスクにより、弾の作成をしています。

弾の発射位置を初期化するため、キャラの座標を弾の座標にコピーしています。

実際の弾の処理は、画面外へ出るまで下方へ向かって進み続ける単純なものです。

CODE_RESET

行動処理をリセットし最初から実行する命令です。

命令位置を示す変数 PC に 0 を代入します。

命令の解説は以上です。1つ1つはとてもシンプルなので、理解はそれほど難しく無いでしょう。

ただ、実際使用するとなると、これらの命令だけでは不足になってくると思います。

その際は、解説を参考に命令を作成、追加していき、独自のスクリプトを作ってみてください。

LIST 3 ―26―1 EX03_script_exec

```
void EX03_script_shot( TCB* thisTCB )
{
    SPRITE* work = (SPRITE*)thisTCB->Work;

    //画面下へ向かって弾の移動
    work->Y += 16;

    if( work->Y > SCREEN_HEIGHT )
    {//画面外に移動したら終了
        TaskKill( thisTCB );
    }else{
    //表示処理
        SpriteDraw( work, 1 );
    }
}
```

```
void EX03_script_exec( EX03_SCRIPT_SHOOT* work )
{
    TCB*        shot_tcb;
    SPRITE*     shot_sprt;

    switch ( work->Code )
    {
        case CODE_END:  //処理が終了していた時はコードと引数データの読み込みを行なう
                work->Code = work->pCode[ work->PC++ ];
                work->Data = work->pCode[ work->PC++ ];
                break;

        case CODE_STOP: //指定の時間停止を行なう
                //停止後、データの読み込み処理へ
                if( --work->Data == 0 ) work->Code = CODE_END;
                break;

        case CODE_LEFT: //一定時間左へ移動する

                work->Sprt.X -= MOVE_SPEED;
                //移動終了後、データの読み込み処理へ
                if( --work->Data == 0 ) work->Code = CODE_END;
                break;

        case CODE_RIGHT://一定時間右へ移動する

                work->Sprt.X += MOVE_SPEED;
                //移動終了後、データの読み込み処理へ
                if( --work->Data == 0 ) work->Code = CODE_END;
                break;

        case CODE_SHOOT://弾を発射する
                //弾の作成
                shot_tcb = TaskMake( EX03_script_shot ,0x2000);
                shot_sprt= (SPRITE*)shot_tcb->Work;
                //弾の初期化
                shot_sprt->X = work->Sprt.X;
                shot_sprt->Y = work->Sprt.Y;

                //発射後、データの読み込み処理へ
```

```
                    work->Code = CODE_END;
                    break;

            case CODE_RESET://リセット処理を行なう
                    work->Code = CODE_END;
                    work->PC   = 0;
                    break;
        }

    }
```

3-27 状態と処理の切り替え

複雑な動作を行なうために

前項で作成したスクリプトでは、いろいろな動作を記述する事が出来ますが、状態をチェック出来るようにはなっていないため、スクリプト上でその動作を切り替える事は出来ません。

しかし、より複雑な動作を行ないたい場合は、こうした処理の切り替えが不可欠になってきます。

そこでここでは、状態によって動作を切り替える方法を紹介します。

処理切り替えの概念

基本的な概念ですが、まず動きのベースとなるスクリプトデータ自体を、1つの動作単位と考えます。

そして、切り替え条件にしたがって、その実行されるスクリプトデータを任意のデータに切り替えてやります。

こうしてやれば、様々な条件で、動作単位ごとに処理を切り替える事が出来ます。

スクリプトデータの切り替え処理

次は実際のプログラムを見ていきます。

まず、サンプルで使用しているスクリプトについてですが、[3－23]で説明した処理を再利用しています。そのため、この部分の処理はそちらを参照してください。

メイン処理

次にメインの処理ですが、これは、一定時間ごとに2種類のスクリプトデータを切り替えて、2種類の動作を行なう物です。

まず、2種類の動作モードのうちどちらの処理かをチェックします。

リストではif文を用いていますが、数が増えるようでしたらswitch文等を用いると良いでしょう。

スクリプトデータの実行

次にモードに合わせて、実行されるスクリプトデータを決定します。

そして、切り替えの条件をチェックし、条件があっていれば処理を切り替えます。

リストでは、モードIDの切り替えと、スクリプト実行データの変更、スクリプト自体のリセット処理を行なう事で実現しています。

また、この時の条件判定はタイマーで行なっていますが、処理内容によってプレイヤーの体力だったり、難易度であっても構いません。

後は、通常の実行時と同様、スクリプトを実行してやれば処理は終了です。

LIST 3 ―27―1 状態と処理の切り替え

```
#define MODE_A 0
#define MODE_B 1
#define CHANGE_TIME_A 200
#define CHANGE_TIME_B 100

typedef struct{
    EX03_SCRIPT_SHOOT    ScriptWork;
    int                  Time;
    int                  ModeID;
} EX03_27_STRUCT ;


void init03_27(TCB* thisTCB)
{
    EX03_27_STRUCT* work = (EX03_27_STRUCT*)thisTCB->Work;

    //使用するテクスチャの読み込み
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥0044.png",&g_pTex[0] );
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥0055.png",&g_pTex[1] );

    //座標と、実行コードの初期化
    work->ScriptWork.Sprt.X = SCREEN_WIDTH  / 2;
    work->ScriptWork.Sprt.Y = SCREEN_HEIGHT / 3;

    work->ScriptWork.Code = CODE_END;
}

void exec03_27(TCB* thisTCB)
{
    EX03_27_STRUCT* work = (EX03_27_STRUCT*)thisTCB->Work;

unsigned char  script_data_A[] =
```

```
{//行動データA
    CODE_LEFT  , 0x18,  //左移動
    CODE_RIGHT , 0x20,  //右移動
    CODE_STOP  , 0x10,  //停止
    CODE_SHOOT , 0x00,  //弾の発射
    CODE_LEFT  , 0x10,  //左移動
    CODE_RIGHT , 0x18,  //右移動
    CODE_SHOOT , 0x00,  //弾の発射
    CODE_LEFT  , 0x18,  //左移動
    CODE_RESET , 0x00,  //行動をリセットし、最初に戻る
};
unsigned char  script_data_B[] =
{//行動データB
    CODE_LEFT  , 0x08,  //左移動
    CODE_SHOOT , 0x00,  //弾の発射
    CODE_RIGHT , 0x08,  //右移動
    CODE_SHOOT , 0x00,  //弾の発射
    CODE_RESET , 0x00,  //行動をリセットし、最初に戻る
};

    //管理用のタイマーを加算
    work->Time++;

    if( work->ModeID == MODE_A )
    {//行動モードA
        //実行するスクリプトデータ
        work->ScriptWork.pCode  = script_data_A;

        if(work->Time >= CHANGE_TIME_A)
        {//条件がそろったら処理を切り替える
            work->ModeID = MODE_B;
            work->ScriptWork.Code = CODE_RESET;
            work->ScriptWork.pCode  = script_data_B;
            work->Time = 0;
        }
    }

    if( work->ModeID == MODE_B )
    {//行動モードB
        //実行するスクリプトデータ
```

```
    work->ScriptWork.pCode  = script_data_B;

    if(work->Time >= CHANGE_TIME_B)
    {//条件がそろったら処理を切り替える
        work->ModeID = MODE_A;
        work->ScriptWork.Code = CODE_RESET;
        work->ScriptWork.pCode  = script_data_A;
        work->Time = 0;
    }
  }

  //スクリプトの実行
  EX03_script_exec( &work->ScriptWork );

  //表示
  SpriteDraw( &work->ScriptWork.Sprt , 0);
}
```

3-28 キャラ選択画面

キャラ選択とは？

ゲームの開始時に自分の好みのキャラクターを選べる機能は最近では珍しくなくなってきました。
特に格闘ゲームでは、登場するキャラクターとほぼイコールでもあるため、この機能がはずせません。

また、各キャラクターには性能差がある事が多く、ゲーム性を広げる意味でも、プレイヤーの選択の幅を広げる意味でも必要な機能といえるでしょう。
ここではこの、「キャラクター選択処理」を、解説してみたいと思います。
とは言うものの、キャラクター選択は様々な種類や手法があるため、とても全てを紹介する事は出来ません。

そこで、シンプルなキャラクター選択の例として、左右で選択カーソルを動かし、決定ボタンで選択を決める処理を紹介します。

キャラ選択のプログラム

キャラの表示処理

では、早速プログラムを見ていきましょう。
まず、選択するキャラクターを表示する位置を決定します。
これは、キャラクターID順に並んでおり、こうする事で、レイアウトをある程度自由に変える事が出来ます。

キャラの選択処理

次にキー入力による、選択キャラクターIDの変更です。
左右で選択IDを増減するようになっており、増減と同時に正規のキャラクターIDの範囲内かどうかを、チェックしています。
ここでは、オーバーしたIDを循環させる事により選択カーソルが、画面端で左右につながる様にしています。

キャラの決定処理

次に決定処理です。これはＺキーで行なっており、サンプルでは決定した時に文字列を変える処理を行なっています。

ここでは省いていますが、実際には決定処理の後に、ゲーム処理への移行を行なう必要があります。

キャラと選択カーソルを表示

最後に、キャラクターと選択カーソルの表示です。

キャラクターは、先ほど定義した配置データを元に表示を行ないます。

同様に選択カーソルは、選択中のキャラクターIDを元に表示座標を決定します。

これらの表示座標は自由に配置できますが、選択処理事態が左右で動かす処理のため、余り離れた座標や入力に合わない座標だと、非常に操作しにくくなってしまいます。

そのため、指定する時はそうならない様に注意して配置してください。

LIST 3 －28－1 キャラ選択画面

```
#define CHR_MAX 4

typedef struct{
    SPRITE          MarkPoint;
    SPRITE          ChrGraph;
    int             SelectID;
    char            DispMess[64];
} EX03_28_STRUCT;

void init03_28(TCB* thisTCB)
{
    //使用するテクスチャの読み込み
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥chr_a.png",&g_pTex[0] );
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥chr_b.png",&g_pTex[1] );
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥chr_c.png",&g_pTex[2] );
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥chr_d.png",&g_pTex[3] );
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥chr_mark.png",&g_pTex[4] );
```

```
}

void exec03_28(TCB* thisTCB)
{
    EX03_28_STRUCT* work = (EX03_28_STRUCT*)thisTCB->Work;
    int loop;
    RECT font_pos = { 0, 0, 640, 480, };
    char str[128];
    int  chr_pos[ CHR_MAX ][2] =
{
    {  144,  240,},
    {  240,  256,},
    {  352,  256,},
    {  448,  240,},
};

    //キー入力によるキャラクター選択
    if( g_DownInputBuff & KEY_RIGHT )
    {
        work->SelectID++;
        //正規IDチェック
        if( work->SelectID >= CHR_MAX )work->SelectID = 0;
    }
    if( g_DownInputBuff & KEY_LEFT  )
    {
        work->SelectID--;
        //正規IDチェック
        if( work->SelectID < 0 ) work->SelectID = CHR_MAX-1;
    }

    if( g_DownInputBuff & KEY_Z  )
    {//キャラクターの決定、ここではメッセージの表示のみ
         switch( work->SelectID )
         {
            case 0: strcpy(work->DispMess,"決定したキャラクターはAです。"); break;
            case 1: strcpy(work->DispMess,"決定したキャラクターはBです。"); break;
            case 2: strcpy(work->DispMess,"決定したキャラクターはCです。"); break;
            case 3: strcpy(work->DispMess,"決定したキャラクターはDです。"); break;
         }
    }
```

```
    //キャラクター選択グラフィックの表示
    for( loop = 0; loop < CHR_MAX; loop++ )
    {
        work->ChrGraph.X = chr_pos[loop][0];
        work->ChrGraph.Y = chr_pos[loop][1];
        SpriteDraw(&work->ChrGraph,loop);
    }

    //選択マークの表示
    work->MarkPoint.X = chr_pos[work->SelectID][0];
    work->MarkPoint.Y = chr_pos[work->SelectID][1];

    SpriteDraw(&work->MarkPoint,4);

    g_pFont->DrawText( NULL, work->DispMess, -1, &font_pos, DT_LEFT,
0xffffffff);

}
```

3-29 隠しプレイヤーキャラを出現させる

隠しキャラクター登場の影響

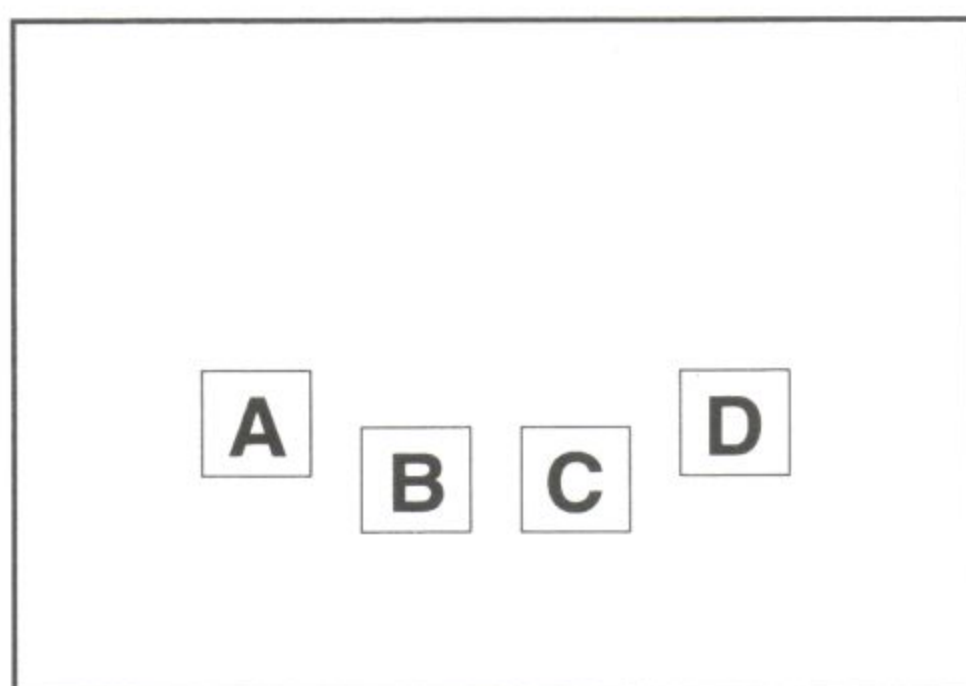
格闘ゲームなどで隠しキャラクターをプログラムに入れる事は、半ば常識に近くなった感があります。

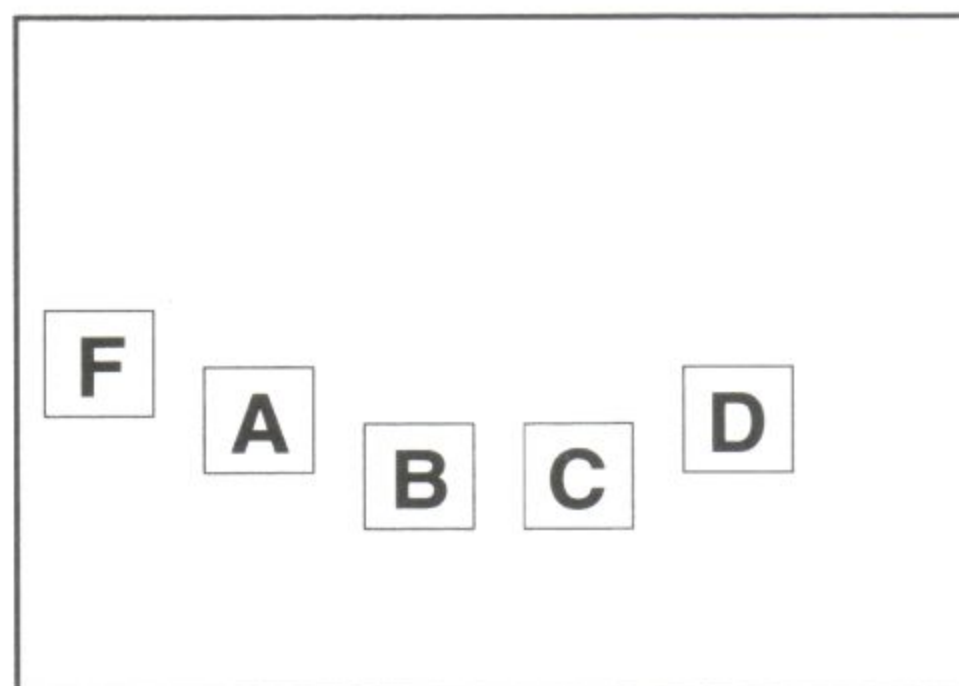
しかし、操作の面から見ると隠しキャラクターの処理は少々面倒です。

何故かというと、隠しキャラクターが出現している時とそうでない時で操作が変わってしまうからです。

そのため、出現時には、その数に合わせてキー操作の処理を変更する必要があります。

図3－29－1 隠しキャラクター出現時で操作が変わるイメージ図

A B C D

通常は、D をカーソル選択中に方向キー右を押せばA にカーソルが移るが

F A B C D

隠しキャラクターが出ている時は、F にカーソルが移る
さらに、追加他のキャラが出てきた場合はまた変更される

隠しキャラクター登場時の処理の変更方法

if 文でチェックする

具体的な手法ですが、大きく分けて2つの手法があります。

1つは、IDの並びを工夫して、隠しキャラとそうでないキャラターを、if 文で1つ1つチェックしてキー操作を変更する手法です。

若干泥臭い手法ですが、処理自体は単純で、隠しキャラクターの数が少ない時によく用いられます。

データの管理でチェックする

もう1つは、出現している時とそうでない時のキー操作を、データとして記録し処理する方法です。

後者は、汎用性が高いのですが比較的処理が面倒です。そのため隠しキャラクターの数が多い時に使われます。

ここでは、手軽な前者の方式を解説していきます。

隠しキャラクターのプログラム解説

ではプログラム見ていきましょう。

サンプルは方向キーの左右でA－Fのキャラクターを選択するようになっています。

EとFのキャラクターは隠しキャラクターになっており、Z、X、それぞれのキーで出現するようになっています。

なお、基本となる処理は[3－28]とほぼ同じですので、違いを見比べてみるのも良いでしょう。

まずは、IDです。IDの並びは以下のようになっており、A～D(0～3)が通常キャラのID範囲でE～F(4～5)が隠しキャラクターのID範囲になっています。

図3－29－2 キャラクターとIDの並び図

0…A	通常キャラ
1…B	通常キャラ
2…C	通常キャラ
3…D	通常キャラ
4…E	隠しキャラ
5…F	隠しキャラ

IDの選択処理そのものは、左右キー入力時に行なわれます。もし、キー入力時にIDが上記の「隠しキャラクターID」であれば専用に処理を行ないます。

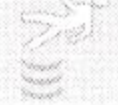
この時の処理はif文でフラグをチェックし、次のIDを選択するだけの単純な物です。

ただ、この時チェックする順番をID順にしておく事で、複数の隠しキャラクターの同時表示に対応できるようにしています。

次に隠しキャラクターの表示フラグですが、こちらは単純にON/OFFを繰り返すだけですので問題は無いでしょう。

最後に表示部分です。こちらも隠しキャラクターのID範囲を利用して表示しています。

表示処理自体は、ループ内でIDと表示フラグを比較し、条件が一致したら表示してやるだけです。

キャラ数を増やす事もさほど難しくないでしょう。

LIST 3 －29－1 隠しプレイヤーキャラを出現させる

```
#define CHR_MAX 6
#define HIDE_CHR_E 4
#define HIDE_CHR_F 5

typedef struct{
    SPRITE          MarkPoint;
    SPRITE          ChrGraph;
    int             SelectID;
    int             HideChr_E;
    int             HideChr_F;
    char            DispMess[64];
} EX03_29_STRUCT;

void init03_29(TCB* thisTCB)
{
    EX03_29_STRUCT* work = (EX03_29_STRUCT*)thisTCB->Work;

    //使用するテクスチャの読み込み
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥chr_a.png",&g_pTex[0] );
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥chr_b.png",&g_pTex[1] );
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥chr_c.png",&g_pTex[2] );
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥chr_d.png",&g_pTex[3] );
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥chr_e.png",&g_pTex[4] );
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥chr_f.png",&g_pTex[5] );
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥chr_mark.png",&g_pTex[6] );
}

void exec03_29(TCB* thisTCB)
{
    EX03_29_STRUCT* work = (EX03_29_STRUCT*)thisTCB->Work;
    int loop;
    RECT font_pos = { 0, 0, 640, 480, };
```

```
    char str[128];
    int  chr_pos[ CHR_MAX ][2] =
{
    {  144,  240,},
    {  240,  256,},
    {  352,  256,},
    {  448,  240,},
    {  536,  224,},
    {  64 ,  224,},
};

    //キー入力によるキャラクター選択
    if( g_DownInputBuff & KEY_RIGHT )
    {
        work->SelectID++;
        if( work->SelectID >= HIDE_CHR_E )
        {//隠しキャラクターが表示されているかチェックし
         //表示されているなら、IDチェックの際に考慮する
            //隠しキャラ非表示なら、そのIDは表示しない
            if( (work->SelectID == HIDE_CHR_E) && !work->HideChr_E )
                work->SelectID++;
            //次のIDも同様
            if( (work->SelectID == HIDE_CHR_F) && !work->HideChr_F )
                work->SelectID++;
        }

        if( work->SelectID >= CHR_MAX )work->SelectID = 0;
    }
    if( g_DownInputBuff & KEY_LEFT  )
    {
        work->SelectID--;
        if( work->SelectID < 0 ) work->SelectID = CHR_MAX-1;
        if( work->SelectID >= HIDE_CHR_E )
        {//減算時も同様、但し逆順にチェックする
            //隠しキャラ非表示なら、そのIDは表示しない
            if( (work->SelectID == HIDE_CHR_F) && !work->HideChr_F )
                work->SelectID--;
            //次のIDも同様
            if( (work->SelectID == HIDE_CHR_E) && !work->HideChr_E )
                work->SelectID--;
```

```
        }
    }

    if( g_DownInputBuff & KEY_X  )
    {//隠しキャラクターEの表示、非表示を切り替える
        work->HideChr_E = !work->HideChr_E;
        work->SelectID = 0;
    }

    if( g_DownInputBuff & KEY_Z  )
    {//隠しキャラクターFの表示、非表示を切り替える
        work->HideChr_F = !work->HideChr_F;
        work->SelectID = 0;
    }

    //キャラクター選択グラフィックの表示
    for( loop = 0; loop < CHR_MAX; loop++ )
    {
        work->ChrGraph.X = chr_pos[loop][0];
        work->ChrGraph.Y = chr_pos[loop][1];

        if( loop >= HIDE_CHR_E)
        {//フラグが立っていた場合のみ、隠しキャラヲ表示する
            if(((loop == HIDE_CHR_E) && work->HideChr_E) |
               ((loop == HIDE_CHR_F) && work->HideChr_F) )
            {
                SpriteDraw(&work->ChrGraph,loop);
            }
        }else{
            SpriteDraw(&work->ChrGraph,loop);
        }
    }

    //選択マークの表示
    work->MarkPoint.X = chr_pos[work->SelectID][0];
    work->MarkPoint.Y = chr_pos[work->SelectID][1];

    SpriteDraw(&work->MarkPoint,6);
}
```

3-30 RPGでの敵の出現

エンカウント処理を考える

RPGでの敵の出現（エンカウント）を考えてみましょう

通常、こういった処理はマップでの移動時に行なわれる事が大半です。

その際大きく分けて、見えている敵に接触した時に戦闘に入るものと、敵が見えていないが一定の確率で即座に戦闘に入るものがあります。

前者は、当たり判定を用いて実装され、後者は通常、乱数を用いて確率の調整を行ないながら実装されます。

前者の手法は多岐にわたり、また当たり判定の手法自体は別項で解説しているので、ここでは後者の手法を解説していきます。

なお、この手法は乱数で行なうため、ランダムエンカウントとも呼ばれます。

ランダムエンカウント

ある程度は出現確率を管理する

さて、このランダムエンカウントですが、実際に乱数といっても、単純に乱数だけだと色々と問題が出てきます。

例えば、戦闘終了直後に、少し移動しただけでまた敵が出てくると、プレイヤーにとってストレスの原因になります。

また、ゲームバランス上でも地形等によって、出現確率や出現する敵の種類を調整できるようにしたい所です。

そのため、こういった出現条件は通常データ化して配列に保存し、乱数確率の決定などはそれらを使用して行ないます。

サンプルでは地形データは３項目あり、以下の様になっています。

EncountStep　最低限移動可能な数少ないと即座に出現する
RandRate　敵の出現確率出現可能になると1/Nで出現する
EnemyLevel　出現する敵の種類

ランダムエンカウントのプログラム

それでは実際のプログラムを見ていきましょう。

サンプルは、Zキーを押すたびに敵の出現条件をチェックし、表示する物です。また、Xキーでチェックする地形条件を変更します。

移動した歩数をチェックする

まずはじめに変数EncountStepをチェックします。この変数は最低限の移動可能数を保証する物で、この数値が0以外の時は敵の出現処理を行ないません。

その場合、このカウンタを1つ減らして終了します。

ランダム判定

次に、チェック可能になった場合ですが、現在の地形データから、乱数の範囲となるデータRandRateを取得します。

そしてその範囲内で乱数を生成し、敵が出現したかどうかをチェックします。ここで、出現確率は、「1/範囲データ」で求められ、乱数の結果が0になった時に出現したとみなされます。

敵の出現処理

その後、敵が出現した場合は、地形データから敵の出現データを取得し、出現処理を行ないます。この時同時に、次に敵が出現するまでの最低移動数を設定しています。あとは、チェックする地形データの変更とメッセージの表示処理です。この部分は特に難しい所は無いでしょう。

LIST 3 －30－1 RPGでの敵の出現

```
#define MAP_ROAD        0
#define MAP_GRASS       1
#define MAP_DESERT      2
#define MAP_MOUNTAIN    3

typedef struct{
    char        EncountStep;
    char        RandRate;
    char        EnemyLevel;
} EX03_30_MAP_DATA;

typedef struct{
    int         EncountStep;
    int         MapDataIndex;
```

```
    char*       MessData;
    int aaa;
} EX03_30_STRUCT;

void exec03_30(TCB* thisTCB)
{
    EX03_30_STRUCT* work = (EX03_30_STRUCT*)thisTCB->Work;
    RECT font_pos = { 0, 0, 640, 480, };
    char str[128];
    int encount_check;
    char* map_name[] = { "道","草原","砂漠","山岳",};

    EX03_30_MAP_DATA map_data[] =
{
    {  8, 10, 0,},      //道
    {  6,  8, 1,},      //草原
    {  4,  5, 2,},      //砂漠
    {  4,  2, 3,},      //山
};

    //キー入力によるエンカウントチェック
    if( g_DownInputBuff & KEY_Z )
    {
        //チェック時に表示メッセージをクリア
        work->MessData = NULL;

        //一定数移動するまでは、敵は出現しない
        if( work->EncountStep == 0 )
        {

            //地形のデータに合わせて乱数を取得し出現確率を計算する
            encount_check = rand() % map_data[work->MapDataIndex].RandRate;

            if( encount_check == 0)
            {//敵の出現処理
                //地形に合わせた敵の出現
                switch( map_data[work->MapDataIndex].EnemyLevel )
                {
                    case MAP_ROAD:    work->MessData = "道の敵が出現しました"; break;
                    case MAP_GRASS:   work->MessData = "草原の敵が出現しました"; break;
```

```
                    case MAP_DESERT:  work->MessData = "砂漠の敵が出現しました"; break;
                    case MAP_MOUNTAIN:work->MessData = "山岳の敵が出現しました"; break;
                }
                //次に出現するまでの最低移動数を設定
                work->EncountStep = map_data[work-
>MapDataIndex].EncountStep;
            }
        } else {
            //エンカウント可能な歩数を一つ減らす
            work->EncountStep--;
        }
    }

    //チェックする地形データの切り替え
    if( g_DownInputBuff & KEY_X  )
    {
        //マップデータは４種類
        if(++work->MapDataIndex == 4) work->MapDataIndex = 0;
    }

    //出現時メッセージの表示
    if(work->MessData != NULL)
        g_pFont->DrawText( NULL, work->MessData, -1, &font_pos, DT_LEFT,
0xffffffff);

    font_pos.top += 16;
    sprintf( str,"現在の地形データ:  %s",map_name[work->MapDataIndex]);
    g_pFont->DrawText( NULL, str, -1, &font_pos, DT_LEFT, 0xffffffff);

    font_pos.top += 32;
    g_pFont->DrawText( NULL,
    "Zキーで敵の出現チェック、Xキーでデータ切り替え",
    -1,
    &font_pos,
    DT_LEFT,
    0xffffffff);
}
```

3-31 簡易スクリプトを作るー アドベンチャー・RPG編1

シナリオ用のスクリプト

アドベンチャーやRPGなどで、短い物や、特にシナリオの分岐を行なわないゲームならば良いのですが、非常に多数のメッセージやフラグを使用する場合は管理がとても大変になります。

そのため、こういった多数の情報を管理する場合、通常は専用の簡易言語を作成して管理します。

ただ、本格的な実装をするとなると、とても複雑になってしまいます。

そこでここでは[3－23]で紹介した、C言語のマクロ機能を利用する手法を元に、その基礎部分を実装、紹介していきます。

シナリオスクリプトの処理概念

まず、処理概念ですが、基本的な部分では、[3－23]で紹介した処理と大きく変わる点はありません。

ただ、先の処理は処理が完全に終了するまで常に実行し続ける「逐次実行型」でしたが、今回の処理は、何らかのイベントが起こった時にはじめて対象のスクリプトデータを実行する、いわゆる「イベント起動型」になっています。

なぜこのような違いがあるかというと、同時に実行しなくてはならない、スクリプトデータの数に差があるためです。

アクションゲームやシューティングでは、基本的に画面上に出現する物体の数だけデータの処理を行ないます。

すなわち、何処でどのような処理を行なうのかの予測がつけやすいのです。

ところが、RPGやアドベンチャーだと、何処でどのようなデータを実行するかは、非常に予測が付けにくくなります。

そのため、仮に遂次実行型でデータを処理すると、何処でイベントが起きても良いように、常に全部のデータを実行しなくてはなりません。これでは、さすがに無駄が多すぎます。

そこで、イベントが起こった時だけ処理を行なう、イベント起動型の処理にする必要があるのです。

図3 －31－1 逐次実行方とイベント起動型の違いのイメージ図

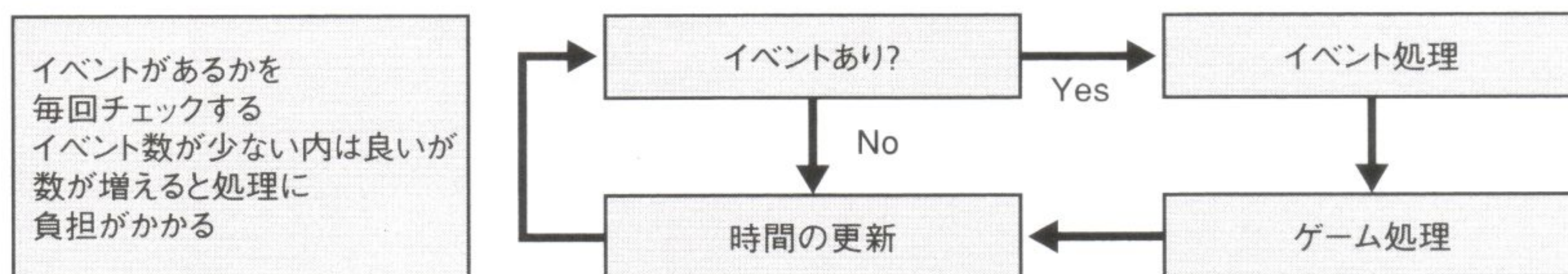
イベント起動型

好きな時に（都合のいい時に）
イベント発生
常に見張る必要が無いので
処理速度は安定する

イベント1	処理1
イベント2	処理2
イベント3	処理3

逐次実行型

イベントがあるかを
毎回チェックする
イベント数が少ない内は良いが
数が増えると処理に
負担がかかる

イベントあり?
Yes → イベント処理 → ゲーム処理 → 時間の更新
No → 時間の更新 → イベントあり?

3-32 簡易スクリプトを作る－アドベンチャー・RPG編2

スクリプトの仕様を考える

それではまず、スクリプトの仕様から考えて行きましょう。

RPG等では、シナリオ進行やアイテムの取得など、基本的にフラグの管理が必須になりますので、まずフラグ設定の命令が必要です。

ここでは最低限の命令として、フラグのON/OFFだけを行なう様にします。

次にフラグをチェックし、条件分岐する命令が必要です。これが無いと、処理を分ける事が出来ません。

こちらも同様に、フラグのON/OFFに合わせて分岐を行なう命令を2種類用意します。

最後に、処理内容を外部に伝える命令です。これは作り方によって千差万別で、関数外部で自動的に取得するようにしている物もあります。

ここではシンプルに、メッセージをバッファに格納する命令を実装しています。外部からはこれを参照してメッセージを表示します。

上記の命令に、処理終了命令を追加して、全部で6種類のスクリプト命令を作成する事にします。

なお、各命令にはそれぞれ、必要な引数データを渡せるようにします。

ただし、処理を単純化するため、命令によらず省略は出来ない（必ず引数を渡す）仕様とします。

引数の内容と命令一覧は以下の通りです。

図3－32－1 命令と、引数データの一覧表

命令		
0x00	CODE_END:	命令が終了終了したことをしめす
0x01	CODE_SET:	引数1のフラグをセットする
0x02	CODE_CLR:	引数1のをフラグをクリアする
0x03	CODE_JUMP_ON:	引数1 のフラグをチェックし設定されていたら、引数2へ分岐する
0x04	CODE_JUMP_OFF:	引数1 のフラグをチェックし設定されていなければ、引数2へ分岐する
0x05	CODE_PRINT:	引数3に渡されたメッセージを表示する

スクリプトの使用方法

次に実際のスクリプトデータの使用法を見ていきます。

サンプルプログラムでは、4つのスクリプトデータを使用しています。

まず初期化時に、初期化用のスクリプト実行しています。

スクリプトデータを見てもらえれば分かると思いますが、これは、初期化時にメッセージを表示するスクリプトです。

次にメインでは、Z、X、上キーをイベントとみなし、3つのスクリプトデータを起動するようにしています。

これは、各キーをある一定の組み合わせで押していく事で、文章を進めるシナリオになっています。

間違ったキーを押すと、正解のキーを押すように促されます。

ここでは、キー入力で代用していますが、実際のRPGではキャラに話しかけたときを想像してもらえるとイメージしやすいでしょうか。

LIST 3 －32－1

```
void init03_script_rpg(TCB* thisTCB)
{
    EX03_SCRIPT_RPG* work = (EX03_SCRIPT_RPG*)thisTCB->Work;

EX03_SCRIPT_RPG_DATA script_data_init[] =
{//初期化データ
    {CODE_PRINT    ,0,0,"これは初期化時に表示されるメッセージです。¥nまずはZキーを押してください
",},
    {CODE_END      ,},
};

    //初期化用のスクリプトで初期化
    EX03_script_rpg_exec( work ,script_data_init );

}

void exec03_script_rpg(TCB* thisTCB)
{
    EX03_SCRIPT_RPG* work = (EX03_SCRIPT_RPG*)thisTCB->Work;
```

```
    RECT font_pos = {  0,  0,640,480,};
//フラグ種類決定
#define FLAG_Z     0
#define FLAG_UP    1
#define FLAG_X     2

EX03_SCRIPT_RPG_DATA script_data_Z[] =
{//Zキーイベントデータ
    {CODE_JUMP_ON ,FLAG_Z, 4},
    {CODE_PRINT   ,0,0,"フラグを立てました。次に方向キーの上を押してください。",},
    {CODE_FLAG_SET,FLAG_Z,},
    {CODE_END     ,},

    {CODE_PRINT   ,0,0,"すでにフラグは立っています。",},
    {CODE_END     ,},
};

EX03_SCRIPT_RPG_DATA script_data_UP[] =
{//方向キー上イベントデータ
    {CODE_JUMP_ON,FLAG_Z, 3},
    {CODE_PRINT   ,0,0,"Zキーがまだ押されていません。¥n先にZキーを押してください。",},
    {CODE_END     ,},

    {CODE_JUMP_ON ,FLAG_UP, 4},
    {CODE_PRINT   ,0,0,"フラグを立てました。¥n最後にXキーを押してください。",},
    {CODE_FLAG_SET,FLAG_UP,},
    {CODE_END     ,},

    {CODE_PRINT   ,0,0,"すでにフラグは立ちましたので、このキーを押す必要はありません。",},
    {CODE_END     ,},
};

EX03_SCRIPT_RPG_DATA script_data_X[] =
{//Xキーイベントデータ
    {CODE_JUMP_ON,FLAG_UP, 3},
    {CODE_PRINT   ,0,0,"このキーを押す条件が満たされていません。¥n上キーを押してみてください。",},
    {CODE_END     ,},

    {CODE_JUMP_ON ,FLAG_X, 4},
    {CODE_PRINT   ,0,0,"最後のフラグを立てました。¥n以上で処理は終了です。",},
```

```
    {CODE_FLAG_SET,FLAG_X,},
    {CODE_END      ,},

    {CODE_PRINT    ,0,0,"すでに処理は終了しました。",},
    {CODE_END      ,},
};



    //各キー入力をイベント代わりにして、スクリプトを実行する
    //Zキーを押した時の処理
    if( g_DownInputBuff & KEY_Z ) EX03_script_rpg_exec( work ,script_data_Z
);
    //Xキーを押した時の処理
    if( g_DownInputBuff & KEY_X ) EX03_script_rpg_exec( work ,script_data_X
);
    //方向キー上を押した時の処理
    if( g_DownInputBuff & KEY_UP) EX03_script_rpg_exec( work ,script_data_UP
);

    //メッセージ文字の表示
    g_pFont->DrawText( NULL, work->MessBuff, -1, &font_pos, DT_LEFT,
0xffffffff);

}
```

3-33 簡易スクリプトを作るーアドベンチャー・RPG編3

簡易スクリプトの実装部分

では、最後に実際のスクリプト実行部分のプログラムを見ていきましょう。

処理関数 EX03_script_rpg_exec は、フラグを管理する構造体と、実行されるスクリプトデータを引数に受け取ります。

この関数は[3 ー 23]で紹介した手法の実行関数と違い、スクリプト実行時の情報は保存していません。

処理の単純化と、常にスクリプトデータの終了まで実行するため保存の必要が無いからです。

ただし、保存するようにすると、様々な仕様拡張が可能になります。もし興味のある方は改良してみると良いでしょう。

実行部分の処理

さて、実行部分そのものですが、これはすでに紹介した手法と同様に、命令と引数データを読み取り、それに対応した処理を行なうというものです。

ただしここでは、データの最後まで実行する必要があるため、処理そのものを無限ループで括っています。

また、命令処理も即時終了するため、命令とデータは 1 ループに 1 度、必ず読み込んでいます。

終了は、終了命令を取得し、処理が終了した時点で、return 文で、直接関数を終了させています。

スクリプトの命令と処理内容

それでは、以下に各命令の簡単な処理内容を解説を示します。

CODE_END

スクリプト処理を終了させる命令です。

命令取得後、return 文で即時処理を終了させています。

CODE_FLAG_SET

指定のフラグの設定をします。

CODE_FLAG_CLR

指定のフラグの消去をします。

フラグのON/OFFを行なう命令です。

フラグエリアは、バイト単位で確保されているので、省メモリ化のため、ビット単位でフラグを操作しています。

1バイト＝8ビットですので、8で割った配列エリアの、該当フラグ(0-7ビット)フラグを設定するようにしています。

なお、サンプルでは、フラグの消去命令は使用していません。

CODE_JUMP_ON

フラグがONの時分岐します。

CODE_JUMP_OFF

フラグがOFFの時分岐します。

フラグの状態を見て分岐する命令です。

条件が成立していたら、引数データの指定する数だけ命令を飛び越して分岐します。

なお、命令終了後自度的に、1命令更新するため、実際の引数データから1引いた値を加算しています。

CODE_PRINT

メッセージの表示を行います。

ただし表示の制御は外部で行なう必要があるため、実際は外部に対してデータのコピーを行なうだけです。

以上で、命令の解説は終了です。

出来る限りシンプルにしたため、機能はかなり限られてしまいました。

そのため、単純なゲームならともかく、このまま使用する事は少し難しいでしょう。

ただ、機能を拡張をするのはそう難しく無いと思いますので、必要な機能があれば適宜追加、改良していってみてください。

LIST 3 －33－1 簡易スクリプトを作る(アドベンチャー・RPG編)

```
typedef struct{
    unsigned char       FlagArea[16];
    char                MessBuff[128];  //文字列格納用のバッファ
} EX03_SCRIPT_RPG;

typedef struct{
    unsigned char       Code;           //実行する命令
    unsigned char       Data1;          //実行する命令の1番目の引数
```

```
    unsigned char       Data2;              //実行する命令の2番目の引数
    char*               Message;            //表示用の文字列
} EX03_SCRIPT_RPG_DATA;

#define CODE_END         0x00
#define CODE_FLAG_SET    0x01
#define CODE_FLAG_CLR    0x02
#define CODE_JUMP_ON     0x03
#define CODE_JUMP_OFF    0x04
#define CODE_PRINT       0x05

#define MOVE_SPEED       8

void EX03_script_rpg_exec( EX03_SCRIPT_RPG* work, EX03_SCRIPT_RPG_DATA*
pCode )
{
    int                 flag_pos;           //フラグの位置
    unsigned char       data1;              //現在実行中の命令の引数
    unsigned char       data2;              //現在実行中の命令の引数
    unsigned char*      pStr;               //表示用文字列へのポインタ

    while( true ){
        //データを読み出す
        data1 = pCode->Data1;
        data2 = pCode->Data2;
        pStr  = pCode->Message;

        switch ( pCode->Code )
        {//スクリプト命令を実行
            case CODE_END:   //スクリプト処理の終了
                return;
                break;

            case CODE_FLAG_SET: //フラグを立てる
                //バイト中のフラグ位置を計算
                flag_pos = data1 & 0x03;
                //フラグセット
                work->FlagArea[ data1 / 8 ]  |= 1 << flag_pos;
                break;
```

```
            case CODE_FLAG_CLR: //フラグを消去
                    //バイト中のフラグ位置を計算
                    flag_pos = data1 & 0x03;
                    //フラグクリア
                    work->FlagArea[ data1 / 8 ]  &= ~(1 << flag_pos);
                    break;

            case CODE_JUMP_ON: //フラグをチェックし、立っていたら処理を分岐させる
                    //バイト中のフラグ位置を計算
                    flag_pos = data1 & 0x03;
                    //フラグのチェックと分岐
                    if( work->FlagArea[ data1 / 8 ]  & (1 << flag_pos) )
pCode += data2-1;
                    break;

            case CODE_JUMP_OFF: //フラグをチェックし、立っていなければ処理を分岐させる
                    //バイト中のフラグ位置を計算
                    flag_pos = data1 & 0x03;
                    //フラグのチェックと分岐
                    if( !(work->FlagArea[ data1 / 8 ]  & (1 << flag_pos)) )
pCode += data2-1;
                    break;

            case CODE_PRINT://メッセージを表示する(外部への情報出力)
                    //実際はバッファにコピーするだけで、表示は呼び出し側で行なう
                    strcpy( work->MessBuff, pStr );
                    break;
        }

        //処理するスクリプトデータを1つ進める
        pCode++;
    }

}
```

Chapter 4

インターフェース

逆引き ゲームプログラミング

Game programming

4-1 Windowsでキーボードを使う

キーボードを使うAPI

Windows ではキーボードを使う代表的な API として GetKeyboardState と、GetAsyncKeyState の２種類のインターフェースがあります。

どちらを使ってもよいのですが、ゲームでは複数のキーを同時に取得するケースが多いため、ここでは GetKeyboardState を使用します。

GetKeyboardState

GetKeyboardState では、キー情報を取得するために256Byte のキー入力用配列変数を用意する必要があります。

取得後、そのキーに対応した配列の最上位ビットの状態を見れば、そのキーの ON/OFF がわかります。

キー情報はまとめておくとあとで使いやすい

ここでは、ゲームでよく使うキーをまとめて１つのグローバル変数 g_InputBuff に格納しています。

キー情報はゲームでは頻繁に参照するので、必要なキーをこうしてまとめておくと置くと、より使いやすくなります。

LIST 4 －1－1

```
//キー情報を取得
GetKeyboardState(g_KeyboardBuff);        //キーボード情報の取得
g_KeyInputBuff = 0;
if( g_KeyboardBuff[VK_UP]    & 0x80 ) g_KeyInputBuff |= KEY_UP;
if( g_KeyboardBuff[VK_DOWN]  & 0x80 ) g_KeyInputBuff |= KEY_DOWN;
if( g_KeyboardBuff[VK_LEFT]  & 0x80 ) g_KeyInputBuff |= KEY_LEFT;
if( g_KeyboardBuff[VK_RIGHT]& 0x80 ) g_KeyInputBuff |= KEY_RIGHT;
if( g_KeyboardBuff[ 'Z' ]    & 0x80 ) g_KeyInputBuff |= KEY_Z;
if( g_KeyboardBuff[ 'X' ]    & 0x80 ) g_KeyInputBuff |= KEY_X;
```

4-2 Windowsでジョイスティックを使う

ジョイスティックを使うAPI

Windows上でジョイスティックを使うにはAPIのjoyGetPosを使用します。

DirectXにはジョイスティック入力を支援するDirectInputもあるのですが、3つ以上のジョイスティックを使うなどの事情が無い限り、こちらを使用するほうがはるかに簡単です。

joyGetPosの使いかた

使用方法ですが、引数として、接続されているジョイスティックのIDと、ジョイスティックの入力状態を格納する構造体、JOYINFOを渡します。

気をつける点として、方向キーXとY、両軸の返り値が、アナログ値であることです。

アナログではないジョイスティックを接続している場合、返り値は上を押した場合が0、下を押した場合が65535となっています。

同様に左右も、左が0、右が65535となります。中央値は両方とも32767です。

この値が入力された時に、それぞれの方向キーをジョイスティック用のバッファに反映させています。

ジョイスティックを調べたいとき

もしジョイスティックの性能や、接続されているかどうかを調べたい時には、joyGetDevCapsを使用するとよいでしょう。

ジョイスティックとキーボードを同時に使う

最後にジョイスティックとキーボードの入力をOR演算子で合成しています。

こうする事で、ジョイスティックとキーボードを等価に扱う事ができます。

LIST 4－2－1

```
//ジョイスティックの情報を取得
g_JoystickBuff = 0;
if( !joyGetPos( JOYSTICKID1, &joyinfo) )
{
    if( joyinfo.wYpos   == 0     ) g_JoystickBuff |= KEY_UP;
    if( joyinfo.wYpos   == 65535 ) g_JoystickBuff |= KEY_DOWN;
```

```
        if( joyinfo.wXpos    == 0     ) g_JoystickBuff |= KEY_LEFT;
        if( joyinfo.wXpos    == 65535 ) g_JoystickBuff |= KEY_RIGHT;
        if( joyinfo.wButtons & 0x01  ) g_JoystickBuff |= KEY_Z;
        if( joyinfo.wButtons & 0x02  ) g_JoystickBuff |= KEY_X;
    }

    //ジョイスティックとキーボードの入力を合成
    g_InputBuff = g_JoystickBuff | g_KeyInputBuff;
```

4-3 Windowsでマウスを使う

マウスの使いかた

Windowsでマウスを使うには、いくつか方法がありますが、標準的なAPIである、GetCursorPosを使うのが、一番手軽でしょう。使用方法も単純で、座標格納用の変数へのポインタを渡してあげれば、現在のマウス座標が戻ってきます。

座標の変換

ただ、注意しなくてはならないのは、戻ってきた座標はWindows上での座標です。

実際に使用するには、クライアントウィンドウの座標へ変換してやらなければなりません。

```
//マウス情報を取得
GetCursorPos( &mouse_pos );
```

座標を変換するには、ScreenToClientを使います。

変換するクライアントウィンドウのウィンドウハンドルと、変換する座標を渡せば座標が変換されます。

```
//クライアント座標へ変換
ScreenToClient( g_hWnd, &mouse_pos );
g_MousePos.x = mouse_pos.x;
g_MousePos.y = mouse_pos.y;
```

マウスボタンの状態の取得

マウスボタンの状態はキー情報の取得と同時に行なっています※。

ここでもそれぞれの情報は、ゲーム全体から参照されやすいよう、グローバル変数g_MousePos、g_MouseButtonに格納しています。

```
g_MouseButton = 0;
if( g_KeyboardBuff[VK_LBUTTON] & 0x80) g_MouseButton |= MOUSE_L;
if( g_KeyboardBuff[VK_RBUTTON] & 0x80) g_MouseButton |= MOUSE_R;
```

※[4－1]Windowsでキーボードを使うを参照。

4-4 入力キーのON/OFFの瞬間の判定

ボタンを押す時、離す時

ゲームでは入力キーのON/OFFの瞬間を知りたくなる事があります。

例えば、ショットを打つ時は、押した瞬間だけの情報が欲しくなりますし、タメ撃ち等の処理を行なうにはボタンを離した瞬間が分からなくてはいけません。

判定の方法

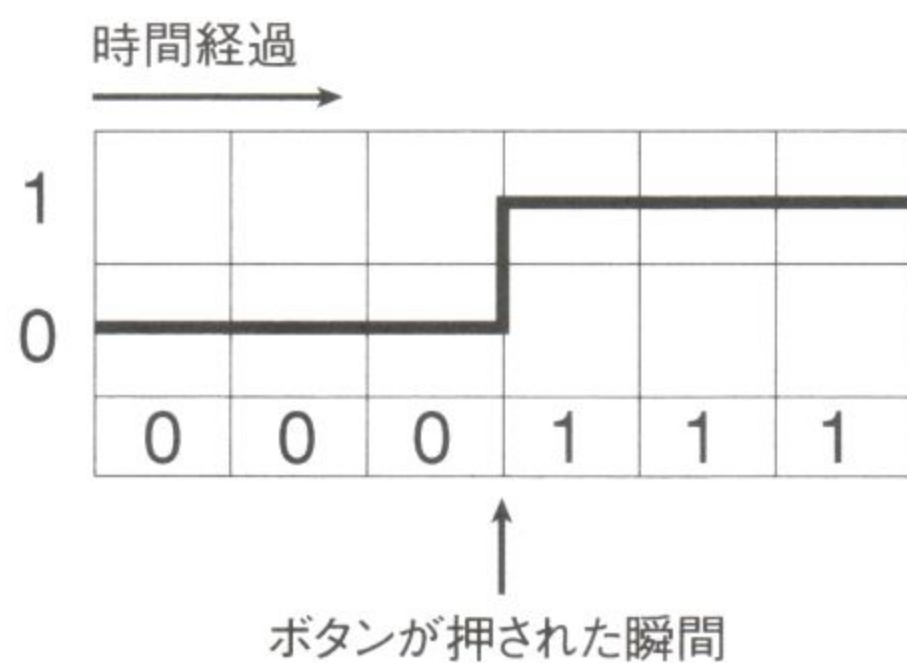
考え方としては、1フレーム前のキー状態を記録しておき、それを現在のキー状態と比較して判定します。

図の真理値表を見てもらえればわかりますが、ONの場合は、ボタンを押した瞬間でかつ前フレームが押されていない時だけフラグがたちます。

同様にOFFの場合は、ボタンを離した瞬間で前フレームが押されていた時だけフラグがたちます。

図4 ―4―1 前フレームと現在のフレームを比較する真理値表

現在だけボタンが押されていれば、ボタンが押された瞬間とみなせる。

前回のキー	現在のキー	結果
0	0	0
0	1	1
1	0	0
1	1	0

論理演算ですっきりさせる

その際、キーを1つ1つif文でチェックしてもいいのですが、論理演算を使うと手軽にすっきりと記述できます。

LIST 4 －4－1

```
//1フレーム前のキー状態を保存
BeforeInputBuff = g_InputBuff;

|
キー入力処理
|

//入力のON/OFFを算出
g_DownInputBuff  =  g_InputBuff & ~BeforeInputBuff;
g_UpInputBuff    = ~g_InputBuff &  BeforeInputBuff;
```

ONの場合は、1フレーム前のキー情報を反転させて、AND演算を行ないます。

OFFの場合はまったく逆で、現在のキー情報を反転させて、1フレーム前のキー情報とAND演算を行ないます。

4-5 タメ撃ちをするには

タメ撃ちとは

シューティング等ではボタンを押している間や離している間に、力をため、ボタンを押す(離す)ことで力を一気に解放する、という演出がよく見受けられます。

ここでは、その「タメ」の実装方法を見ていきましょう。

タメ撃ちの判定

"タメる"処理

まずはタメ撃ちからです。最初に、タメるためのボタンが押されているかをチェックします。

ボタンが押されている場合はパワーをため、タスクワーク上のaccmulate_power_onに格納します。

もし、ここでパワーの上限までタメていたら、そこでパワーを上限に合わせます。

どこまでタメられるかは、MAX_POWERで定義しています。

LIST 4－5－1

```
#define MAX_POWER   100
    EX04_05_STRUCT* work = (EX04_05_STRUCT*)thisTCB->Work;
    //Xキーを押している間はタメ
    if(g_InputBuff & KEY_X )
    {
        work->accmulate_power_on++;
        if( work->accmulate_power_on > MAX_POWER )
        {
            work->accmulate_power_on = MAX_POWER;
        }
    }
```

ボタンが離された時の処理

次に、タメるためのボタンが離されたかをチェックします。

LIST 4 ー5ー2

```
//離した瞬間のみ処理する
if(g_UpInputBuff & KEY_X )
{
    if( work->accmulate_power_on >= MAX_POWER)
    {
        TaskMake(exec04_05_Success,0x2000);
    }
    work->accmulate_power_on = 0;
}
```

ボタンが離された瞬間に、パワーが最大値までたまっていればその処理を行ないます。
ここではその処理として、パワーが解放された事を表示するタスクを生成しています。
もしパワーが足りてない場合は処理を行なわず、今までタメていたパワーを0に戻します。

4-6 ボタンを離している間にタメるには

押してタメるのとは逆の処理

ボタンを離している間にタメる処理も見てみましょう。

基本的な構造はほぼ同じです。

ボタンを離している時にパワーを蓄積し、押した瞬間にパワーの解放に関する処理を行ないます。

使いかたに注意する

ただ、ボタンを離した時にタメているため、実質的に「自動タメ」と同じ事になります。

こういった形で、自動的に処理を行なう場合、ゲームを遊ぶ人が混乱しない様、扱いには注意が必要です。

LIST 4 －5－3

```
//Zキーを離している間はタメ
if(!(g_InputBuff & KEY_Z))
{
      work->accmulate_power_off++;
      if( work->accmulate_power_off > MAX_POWER )
      {
           work->accmulate_power_off = MAX_POWER;
      }
}

//押した瞬間のみ処理する
if(g_DownInputBuff & KEY_Z )
{
     if( work->accmulate_power_off >= MAX_POWER)
     {
          TaskMake(exec04_06_Success,0x2000);
     }
     work->accmulate_power_off = 0;

}
```

4-7 ボタンの同時押しの判定

判定には余裕を持たせる

ゲームではボタンの同時押しをして緊急回避する…といった、行動はよく見かけます。

ただ、実際に同時押しをプログラムしてみると、なかなか同時押しを受け付けてくれません。

これは、1フレームで同時にキー入力をしなくてはいけないため、入力のタイミングが厳しすぎて起る現象です。

そのため、実際には数フレーム前までのキー状態を保持しておき、一定時間内でボタンが同時に押されたかをチェックする必要があります。

判定のプログラミング

履歴を保存する

判定の方法ですが、まず履歴を保持するための履歴バッファを作り、それを毎フレーム更新していきます。

更新は要素を古い順に1つづつコピーしていく事で行ないます。

LIST 4 ―7―1

```
#define INPUT_HISTORY_COUNT 16
typedef struct{
    SPRITE          sprt;
    float           AddX;
    float           AddY;
    unsigned char   HistoryInputBuff[INPUT_HISTORY_COUNT];
} EX04_07_STRUCT;

    EX04_07_STRUCT* work = (EX04_07_STRUCT*)thisTCB->Work;

    int same_button_check = 0;
    int loop;

    //キー状態の履歴を更新
    for( loop = INPUT_HISTORY_COUNT-1; loop>0; loop--)
    {
```

```
        work->HistoryInputBuff[loop] = work->HistoryInputBuff[loop-1];
    }
    //最新のキー押下状態を保存
    work->HistoryInputBuff[0] = g_InputBuff;            //1フレーム前のキー状態を保存
```

ボタンの判定

その後、or 演算子を使い、過去に押されたボタンを1つにまとめます。(ここでは、過去4フレーム分の状態をまとめています)

その後、押されたボタンをチェックすればOKです。

もし、同時に押す時間が短いと感じたら、チェックするフレームを長めに取ってみてください。

LIST 4 −7−2

```
    //過去4フレーム分の履歴をチェック
    for(loop=0; loop<4; loop++)
    {
        same_button_check |= work->HistoryInputBuff[loop];
    }

    //同時押し成功
    if( same_button_check & ((1 << KEY_X)|(1 << KEY_Z)) )
    {
        TaskMake(exec04_07_Success,0x2000);
    }
```

図4 −7−1 前フレームと現在のフレームを比較する真理値表

ボタンA	OFF	OFF	ON	OFF	ON	OFF	OFF
ボタンB	OFF	OFF	ON	OFF	OFF	OFF	OFF
判定	×	×	×	×	×	×	×

現在の入力だけチェックすると、
少し入力が遅れただけで判定が出来ず、厳しい

ボタンA	OFF	OFF	ON	OFF	ON	OFF	OFF
ボタンB	OFF	OFF	ON	OFF	OFF	OFF	OFF
判定	×	×	×	×	○	×	×

過去の入力にもチェックすることで、
入力に余裕を持たせることが出来る

4-8 連射をするには

自分で連射するか機能にするか

ゲームと連射は、切っても切り離せない関係にあるようです。

中でもシューティング系統においては、連射機能の有無でゲームバランスが変わってしまうゲームが大半を占めています。

そういった背景もあり、最近では必死にボタンを連打するゲームはめっきり減ってしまいました（個人的にはこういったゲームは嫌いではないのですが）。

連射機能

ソフトウェアによる連射機能は、「ボタンを押しっぱなしにしている間、一定時間毎にON/OFFの状態を切り替える」事で実現します。

連射の速度変更は、ON/OFFの間隔を変える事で行ないます。

連射機能を作る

まず、連射に該当するキーが押されているか判定します。

キーが押されていた場合、連射間隔のカウンタRapidCountに加算します。

RapidCountは、フレーム間を越えて保持される必要があるためタスクワーク上に設けています。

もしこの値が一定数を越えていたら、その瞬間だけボタンをONになったものとして扱います。

あとは、これを連射をするキーの数だけ繰り返せばOKです。

このリストでは、コードの単純さを上げるためループを展開していますが、連射するキーが大量にある場合はfor文を使用したほうがよいでしょう。

ボタンを押していないときの処理

最後に押されていなかった場合には、連射間隔のカウンタを0にする事を忘れないようにしてください。

ボタンを押すたびに連射の間隔がずれてしまいます。

LIST 4 －8－1

```
//連射等特殊処理
g_RapidBuff = 0;
g_HalfRapidFlag  = 0;
if( g_HalfRapidFlag ){
    if( g_InputBuff & KEY_X )
    {//Xボタンが押されている時のみ連射処理
        ++pTASK_work->RapidCount[ KEY_INDEX_X ] %= RAPID_COUNT;
        if( !pTASK_work->RapidCount[ KEY_INDEX_X ] )
        {
            g_RapidBuff |= KEY_X;
        }
    } else {
        pTASK_work->RapidCount[ KEY_INDEX_X ] = 0;
    }

    if( g_InputBuff & KEY_Z )
    {//Zボタンが押されている時のみ連射処理
        ++pTASK_work->RapidCount[ KEY_INDEX_Z ] %= RAPID_COUNT;
        if( !pTASK_work->RapidCount[ KEY_INDEX_Z ] )
        {
            g_RapidBuff |= KEY_Z;
        }
    } else {
        pTASK_work->RapidCount[ KEY_INDEX_Z ] = 0;
    }
}
```

一定時間だけ連射をする処理

基本的な構造は自動連射の時と大きくは変わりません。

ただ、ボタンを押して一定時間だけ連射を行なうためのカウンタHalfRapidCountを新規に設けています。

このカウンタが有効な間だけ連射を行なうようにします。

気を付けなくてはならないのは、「ボタンが押されたら、一定間隔プレイヤーの意思とは無関係に連射を行なう」という事です。

問題になる事は少ないですが、状況によってはバグとも取られかねないので注意しましょう。

LIST 4 −8−2

```
//半自動連射
        //ボタンが押されていたら半自動連射モードに切り替え
        if( g_DownInputBuff & KEY_X )
            pTASK_work->HalfRapidCount[ KEY_INDEX_X ] = HALF_RAPID_COUNT;

        //半自動連射モード中は連射
        if( pTASK_work->HalfRapidCount[ KEY_INDEX_X ] != 0)
        {
            ++pTASK_work->RapidCount[ KEY_INDEX_X ] %= RAPID_COUNT;
            if( !pTASK_work->RapidCount[ KEY_INDEX_X ] )
            {
                g_RapidBuff |= KEY_X;
            }
        } else {
            pTASK_work->RapidCount[ KEY_INDEX_X ] = 0;
        }

        if( pTASK_work->HalfRapidCount[ KEY_INDEX_X ] )
                pTASK_work->HalfRapidCount[ KEY_INDEX_X ]--;


        //同様にZキーも処理する
        if( g_DownInputBuff & KEY_Z )
            pTASK_work->HalfRapidCount[ KEY_INDEX_Z ] = HALF_RAPID_COUNT;

        if( pTASK_work->HalfRapidCount[KEY_INDEX_Z] != 0)
        {
```

```
        ++pTASK_work->RapidCount[ KEY_INDEX_Z ] %= RAPID_COUNT;
        if( !pTASK_work->RapidCount[ KEY_INDEX_Z ] )
        {
            g_RapidBuff |= KEY_Z;
        }
    } else {
        pTASK_work->RapidCount[ KEY_INDEX_Z ] = 0;
    }

    if( pTASK_work->HalfRapidCount[ KEY_INDEX_Z ] )
            pTASK_work->HalfRapidCount[ KEY_INDEX_Z ]--;
```

4-9 同時方向２回押しによるダッシュの入力判定

２回連続押し

ゲームで同方向に２回入力する事を判定する事はよくあります。

ほとんどがダッシュ等の移動に関する入力判定ですが、攻撃を行なったりするケースもあり、簡易なコマンド入力といえるでしょう。

判定のアルゴリズム

判定のアルゴリズムですが、方向キーが押された瞬間を検知し、そこから一定時間内にもう一度入力があるかをチェックします。

一定時間内に入力がなされていたら、入力が成功したとみなします。

これを各方向について行なうだけです。今回のプログラムでは２ヶ所ですが、状況に応じて変更するとよいでしょう。

LIST 4 －９－１ 同方向２回押しによる入力判定

```
typedef struct{
    unsigned char   KeyTime[2];
} EX04_09_STRUCT;

void exec04_09(TCB* thisTCB)
{
#define INPUT_COUNT 12
#define KEY_INDEX_LEFT  0
#define KEY_INDEX_RIGHT 1
    EX04_09_STRUCT* work = (EX04_09_STRUCT*)thisTCB->Work;

    if( g_DownInputBuff & KEY_LEFT )
    {
        //一定時間内にキーがもう一度入力されていたら入力成功
        if( work->KeyTime[ KEY_INDEX_LEFT ] != 0)
        {
            TaskMake(exec04_09_Success,0x2000);
        }else{
            work->KeyTime[ KEY_INDEX_LEFT ] = INPUT_COUNT;
```

```
            }
        }

        if( g_DownInputBuff & KEY_RIGHT )
        {
            if( work->KeyTime[ KEY_INDEX_RIGHT ] != 0)
            {
                TaskMake(exec04_09_Success,0x2000);
            }else{
                work->KeyTime[ KEY_INDEX_RIGHT ] = INPUT_COUNT;
            }
        }
        //連続入力時間のカウント
        if( work->KeyTime[ KEY_INDEX_LEFT  ] != 0 )
            work->KeyTime[ KEY_INDEX_LEFT  ]--;
        if( work->KeyTime[ KEY_INDEX_RIGHT ] != 0 )
            work->KeyTime[ KEY_INDEX_RIGHT ]--;
    }
```

4-10 コマンド入力を考える

コマンド入力とは

格闘ゲームなどでおなじみのコマンド入力は、最近ではシューティング等にも使用されたりと、色々使い道が広がっているようです。

実際にコマンド入力を実現する手法は幾つかあるのですが、今回は応用範囲が広いバッファを用いた方法を紹介します。

バッファを使ったコマンド入力判定

履歴バッファを用意

基本的な考え方ですが、まず過去のキー入力を覚えておくための、履歴バッファを用意します。

このバッファにはコマンドとは無関係に、一定期間分のキー入力の履歴を記録しておきます。

コマンド入力完了のチェック

次に、プレイヤーがコマンド入力を完成した事を示す、トリガーをチェックします。

格闘ゲームで言えば、コマンド入力後のパンチボタンやキックボタンがこれに該当します。

キー入力の判定

トリガーが入力されたら、押した瞬間のフレームから遡って、過去のキー入力状態をチェックします。

もしコマンドに該当するキーパターンがあれば、コマンド成功とみなします。

図4－10－1

キー入力履歴バッファに入力データを必要な時間分だけ記録する

左
上
右
下
左
⋮
上
右
下
左

履歴がキーパターンと一致していたらコマンド入力成功

履歴	キーパターン
上	
下	← 右下
右下	← 右下
右下	
右	← 右
下	
下	

4-11 コマンド入力判定プログラム

判定の方法をプログラミング

実際にコマンド入力部分を実装してみます。

履歴を更新／保存

まず、履歴バッファへキー入力の履歴を更新、保存しています。

履歴バッファは、各フレーム間をまたがって記録されるため、static か、タスクワーク上に確保します。

また、この際ボタン情報は保持せず、方向キーのみを保存しています。

LIST 4 －11－1

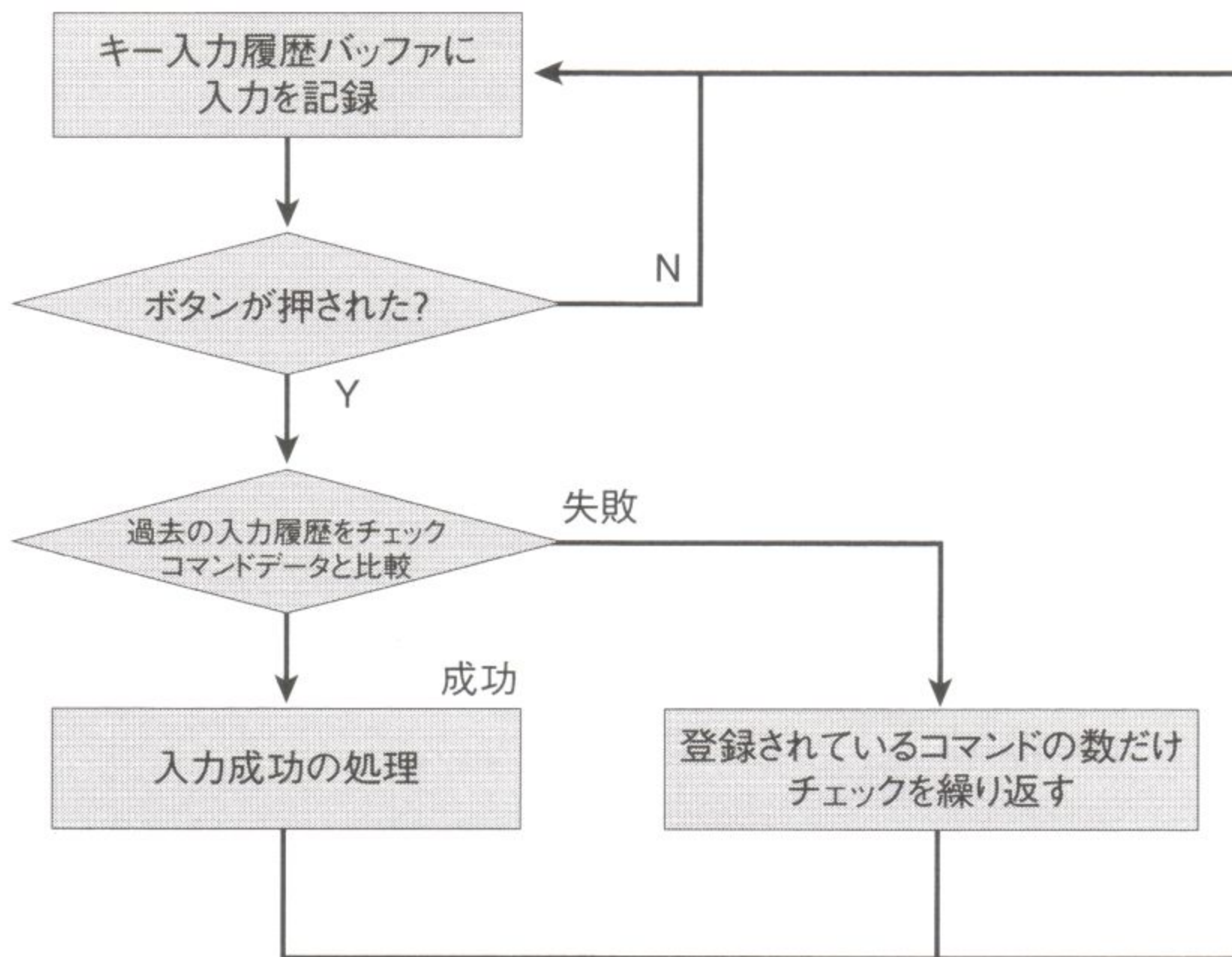
```
//キー状態の履歴を更新
for( loop = HISTORY_COUNT-1; loop>0; loop--)
{
    work->HistoryBuff[loop] = work->HistoryBuff[loop-1];
}
```

判定

次にコマンド入力判定のトリガーをチェックします。ここではＺキーを押した瞬間をトリガーとしています。

トリガーが押されたら、過去の履歴をコマンドの入力データと比較しチェックします。

コマンドのデータは順に、入力受付時間、コマンドのキーパターン（7 つまで）となっており、入力方向に0 を入れることでキーパターンが終了します。

図4 ―11―1 キーパターンデータのデータ構造

チェックの順序に気をつける

キーパターンをチェックする際、より古い方からチェックしていくことに、注意してください。こうしないと、入力の順序が逆になってしまいます。

キーパターンが最後までチェックされれば、コマンドの入力が成功した事になります。

もし、最後までチェックしなかった場合は、そのコマンドは入力されておらず、次のキーパターンデータのチェックに移行します。

LIST 4 ―11―2　コマンド入力

```
void exec04_11(TCB* thisTCB)
{
#define HISTORY_COUNT 100

typedef struct{
    unsigned char   HistoryBuff[HISTORY_COUNT];
} EX04_11_STRUCT;

typedef struct{
    unsigned char   Time;
    unsigned char   Key[7];
} EX04_11_DATA;
```

```
EX04_11_DATA command_data[] =
{
    //コマンド1 下 右下 右
    { {30,},{ KEY_DOWN,  KEY_DOWN|KEY_RIGHT, KEY_RIGHT, 0,}, },
    //コマンド2 右、下、左下
    { {30,},{ KEY_RIGHT, KEY_DOWN,  KEY_DOWN|KEY_RIGHT, 0,}, },
    //コマンド3 下、左下、左
    { {30,},{ KEY_DOWN,  KEY_DOWN|KEY_LEFT,  KEY_LEFT,  0,}, },
    //コマンド終了
    { { 0,},{0,},},
};

    EX04_11_STRUCT* work = (EX04_11_STRUCT*)thisTCB->Work;
    TCB* newTCB;
    int check_command = 0;
    int loop;
    unsigned char* key_data;

    FontPrint(192,160,"INPUT COMMAND");

//キー状態の履歴を更新
    for( loop = HISTORY_COUNT-1; loop>0; loop--)
    {
        work->HistoryBuff[loop] = work->HistoryBuff[loop-1];
    }

    //最新のキー状態を保存
    work->HistoryBuff[0] = g_InputBuff &
(KEY_UP|KEY_DOWN|KEY_LEFT|KEY_RIGHT); //方向キーのみ保存

    //Zキーが押された時に、コマンドの解析を開始
    if( g_DownInputBuff & KEY_Z )
    {
        //チェックするコマンドが終了するまでループ
        while( command_data[ check_command ].Time )
        {
            //履歴上のキー入力パターンをチェック
            key_data = command_data[ check_command ].Key;
            for(loop = 0;loop < command_data[ check_command ].Time; loop++)
            {
```

```
                if( *key_data == work->HistoryBuff[ command_data[
check_command ].Time - 1 - loop] )
                {   //全部のキーパターンが入力されていたらコマンド入力成功
                    key_data++;
                    if(*key_data == 0)break;
                }
            }
            //コマンド入力成功なら、処理を行いチェック終了
            if(*key_data == 0)
            {
                //コマンドを表示するタスクを生成
                newTCB = TaskMake(exec04_11_Success,0x2000);
                newTCB->Work[0] = check_command;
                break;
            }
            check_command++;

        }
    }
}
```

4-12 プレイヤー名前登録

ジョイスティックによる名前入力

ゲーム上での名前入力処理を作成してみましょう。

通常、Windows上での入力はキーボードで行なわれます。

しかし、ゲームでの入力はジョイスティックを使う物が多く、また入力も方向キーとボタンだけという場合が殆どです。

そのため、もしゲーム中に名前の入力をキーボードで行なう場合、はユーザーへ入力機器の変更を促す事になり、インターフェースの統一の面から見ると、余りよい事とはいえません。

そこで統一したインターフェースのために、方向キーとボタンのみで名前入力を行なう処理を作成してみましょう。

もっとも名前入力は、ゲームによって千差万別で、非常に多岐にわたります。

ここでは、もっともシンプルな、方向キーの左右でアルファベットのスクロールによる選択、ボタンで入力の決定とキャンセルを行なう処理を作成します。

扱う文字について

それでは実際の処理を解説していきます。

まずは扱う文字についてです。名前入力に使用される文字は、アルファベットであったりカナ文字だったりする上に、表示の順番や配置等はバラバラです。

そこで、これらを統一的に扱うようにするため、文字の入力時はIDで処理を行ない、実際の文字は変換テーブルを用いて文字の決定に変換するようにしてやります。

こうする事で、文字の表示順番や配置などを気にせずに処理する事ができます。

文字入力のプログラム

次に実際のプログラムです。初期処理終了後、まずはキー入力による文字の選択を行なっています。

ここでは純粋に、選択中の文字IDの切り替え処理のみを行なっています。また入力処理後、選択中のIDが指定の範囲内に収める処理も同時に行なっています。

選択用文字の表示

次は、表示する選択用の文字列です。画面では11個の文字を表示し、それを左右に移動させて選択文字を決定するようになっています。

はじめに選択中の文字から、表示を開始する文字を算出します。

選択文字は表示文字列の中央の文字なので、表示文字数を2で割った値を選択した値から引いてやる事で表示開始の文字を得る事ができます。

その後、得られた開始文字から1文字づつ、文字の表示を行ないます。その際の表示位置は、ループに合わせて表示幅ずつずらしてやる事で行なっています。

表示文字そのものは、IDで管理しているため、表示関数FontPrintで表示する際に、変換テーブルalphabet_dataを用いてIDをアルファベットに変更しています。

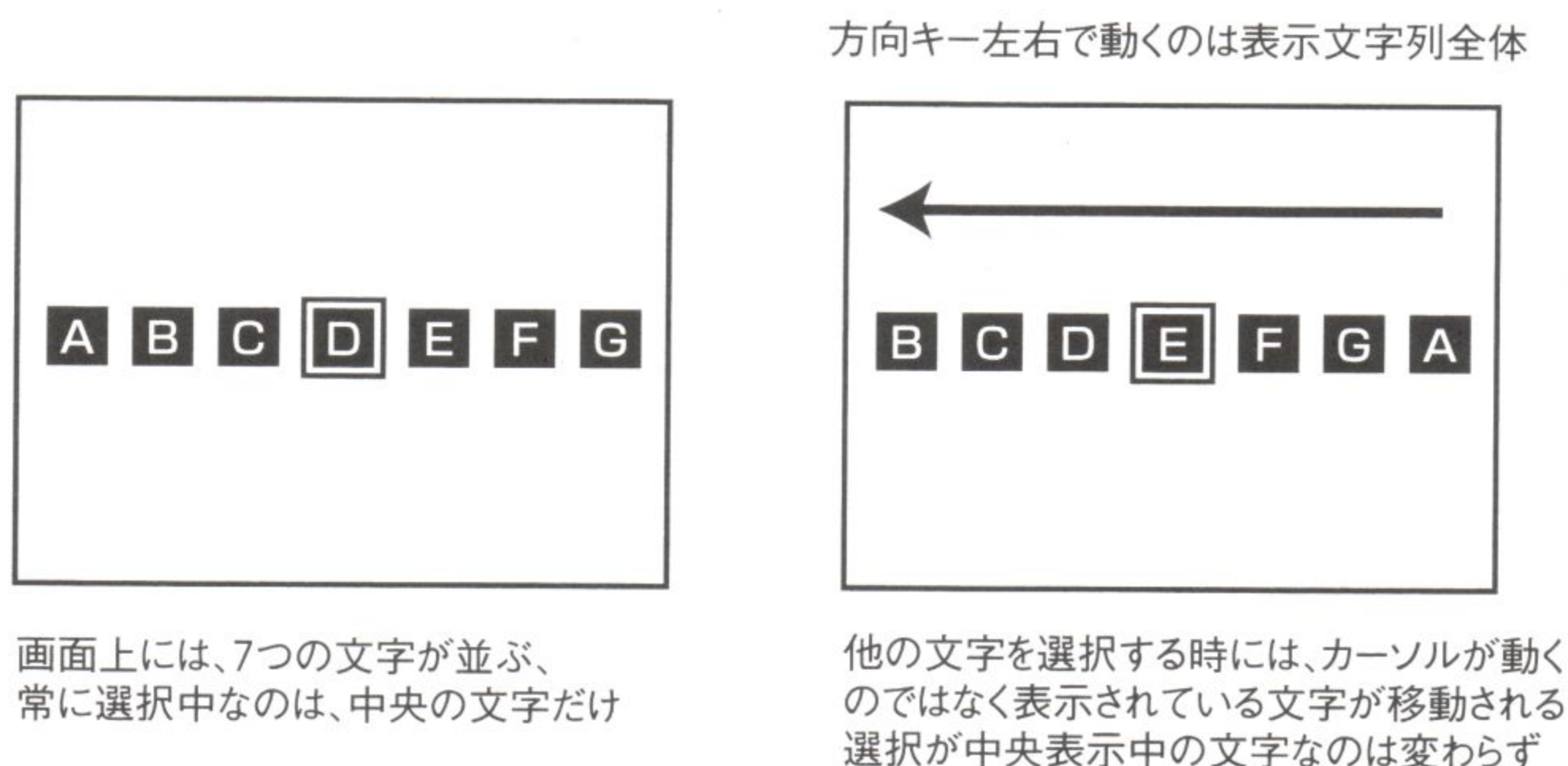
図4－12－1

方向キー左右で動くのは表示文字列全体

A B C D E F G

画面上には、7つの文字が並ぶ、
常に選択中なのは、中央の文字だけ

B C D E F G A

他の文字を選択する時には、カーソルが動く
のではなく表示されている文字が移動される
選択が中央表示中の文字なのは変わらず

文字の決定

次は、文字の決定処理です。

こちらの方は、決定のボタンが押された瞬間に、現在選択中の文字を名前の格納バッファにコピーしてやるだけです。

またこの際、同時に決定した文字数を記録しておき、規定文字数の入力が終了したら、終了処理も同時に行ないます。

ここでは、特に終了処理自体を行なわず、名前のみの表示処理に処理を切り替えています。

キャンセル処理

次はキャンセル処理ですが、こちらの方はシンプルで、決定した文字数を1文字減らし、最後に入

力した文字をNULLで消去するだけです。

ただ、文字数を減らす時に、0以下にならないように注意してください。

最後に決定した名前の表示と、選択マークの表示を行ない、処理は終了です。

LIST 4 －12－1 プレイヤー名前登録

```
#define ALPHABET_NUM      40
//入力可能な名前の文字数
#define NAME_MAX          5
//表示する文字数(奇数にする事)
#define DISP_ALPHABET     11
//文字の表示座標等
#define SELECT_NAME_X     SCREEN_WIDTH / 2
#define SELECT_NAME_Y     SCREEN_HEIGHT/ 2
#define NAME_X            (SCREEN_WIDTH / 2 - 32)
#define NAME_Y            (SCREEN_HEIGHT/ 2 + 64)
#define SELECT_MARK_X     SELECT_NAME_X - 8
#define SELECT_MARK_Y     SELECT_NAME_Y - 8

#define DISPLAY_WIDTH     32

typedef struct{
    SPRITE  Mark;
    int     SelectAlphabetID;
    int     NameCount;
    //入力した名前、終端文字用に+1
    char    Name[NAME_MAX + 1];
} EX04_12_STRUCT;

void EX04_12_name_display(TCB* thisTCB)
{
    EX04_12_STRUCT* work = (EX04_12_STRUCT*)thisTCB->Work;

    //決定した名前の表示
    FontPrint(NAME_X,NAME_Y,work->Name);
}

void init04_12(TCB* thisTCB)
{
    EX04_12_STRUCT* work = (EX04_12_STRUCT*)thisTCB->Work;
```

```
    //使用するテクスチャの読み込み
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥mark32x32.png",&g_pTex[0] );

    //選択マークの表示座標の初期化
    work->Mark.X = SELECT_MARK_X;
    work->Mark.Y = SELECT_MARK_Y;
}

void exec04_12(TCB* thisTCB)
{
    EX04_12_STRUCT* work = (EX04_12_STRUCT*)thisTCB->Work;
    int loop;
    int display_alphabet;
char* alphabet_data[ ALPHABET_NUM ] =
{//使用する文字と、その並びのデータ
    "A", "B", "C", "D", "E", "F", "G", "H",
    "I", "J", "K", "L", "M", "N", "O", "P",
    "Q", "R", "S", "T", "U", "V", "W", "X",
    "Y", "Z", " ", "0", "1", "2", "3", "4",
    "5", "6", "7", "8", "9", "0", "!", "?",
};

    //文字の選択
    if( g_DownInputBuff & KEY_RIGHT  )
    {//選択文字を範囲内に収める
        work->SelectAlphabetID++;
        if(work->SelectAlphabetID >= ALPHABET_NUM)
            work->SelectAlphabetID -= ALPHABET_NUM;
    }
    if( g_DownInputBuff & KEY_LEFT )
    {//選択文字を範囲内に収める
        work->SelectAlphabetID--;
        if(work->SelectAlphabetID < 0)
            work->SelectAlphabetID += ALPHABET_NUM;
    }

    //選択用文字列の表示
    //初めに表示開始文字を決定
    display_alphabet = work->SelectAlphabetID - (DISP_ALPHABET-1) / 2;
```

```
	//同様に範囲内に収める
	if(display_alphabet < 0)
		display_alphabet += ALPHABET_NUM;

	for( loop = 0; loop < DISP_ALPHABET; loop++ )
	{
		FontPrint(
			//表示位置の計算
			SELECT_NAME_X - ((DISP_ALPHABET-1) / 2 * DISPLAY_WIDTH) + loop *
DISPLAY_WIDTH,
			SELECT_NAME_Y,
			alphabet_data[ display_alphabet ]
		);

		//表示する文字を更新する
		display_alphabet++;
		if(display_alphabet >= ALPHABET_NUM)
			display_alphabet-= ALPHABET_NUM;
	}

	if( g_DownInputBuff & KEY_Z )
	{//文字の決定
		//決定した文字を名前文字列にコピー
		strcpy( &work->Name[ work->NameCount ],alphabet_data[ work-
>SelectAlphabetID ] );

		work->NameCount++;
		if(work->NameCount >= NAME_MAX )
		{//規定の入力数を越えたら終了
			//名前入力の終了処理
			//ここでは、名前だけを表示する処理に切り替えています
			TaskChange( thisTCB,EX04_12_name_display );
		}
	}

	if( g_DownInputBuff & KEY_X )
	{//文字のキャンセル
		//決定した文字を１文字もどす
		work->NameCount--;
		//ただし０以下にはしない
```

```
        if(work->NameCount < 0 ) work->NameCount = 0;
        //文字の消去
        work->Name[ work->NameCount ] = NULL;
    }

    //決定した名前の表示
    FontPrint(NAME_X,NAME_Y,work->Name);

    //選択マークの表示（位置は固定
    SpriteDraw( &work->Mark, 0);
}
```

Chapter 5

グラフィックの表示

逆引き ゲームプログラミング

Game Programming

5-1 キャラを表示する

ビットマップの表示

まずは基本中の基本、ビットマップの表示からです。

キャラクター等の表示ができないと、ゲームはまず成り立ちません。

とはいっても、DirectX上ではさほど難しくなく、手軽にビットマップの表示ができます。

使用するAPIはD3DXSPRITEのDrawメソッドです。

データの読み込み

まずは、表示するビットマップデータをテクスチャとして読み込みます。

```
    //使用するテクスチャの読み込み
    if( FAILED( D3DXCreateTextureFromFile( g_pD3Ddevice, "..¥¥0011.png",
&g_pTex ) ) )
    {
        return(false);
    }
```

ビットマップを読み込むのは最初の一度だけなので、初期化時に呼び出します。

g_pD3Ddevice は、データを使用するDirectXデバイスへのポインタ、g_pTexはテクスチャ管理用のオブジェクトへのポインタです。

座標を設定

次に、表示するビットマップの座標をD3DXVECTOR3構造体に設定します。

```
  D3DXVECTOR3 pos;

    pos.x = 320;
    pos.y = 240;
    pos.z = 0;
```

表示

あとはDrawメソッドの引数に、表示するビットマップデータと座標が格納されているポインタを指定してやれば、指定した座標にビットマップグラフィックが表示されます。

ここで、使用しているg_pSpはD3DXSPRITEオブジェクトへのポインタです。

```
    g_pSp->Draw( g_pTex,NULL,NULL,&pos,0xffffffff);
```

D3DXSPRITEは非常に多機能です。

表示に関して、とても多様な設定を行なうことができますが、詳細に関しては他の項目に譲ります。

なお、この表示の機能は非常に良く使うので、関数にしておくと非常に便利です。

本書のサンプルでも、下記のようにまとめた関数を使用しています※。

```
void SpriteDraw(SPRITE* pspr,int bitmap_id)
{
    D3DXVECTOR3 pos;

    pos.x = pspr->X;     //座標の設定
    pos.y = pspr->Y;
    pos.z = 0;

    //スプライトの描画
    g_pSp->Draw( g_pTex[bitmap_id],pspr->SrcRect,NULL,&pos,0xffffffff);
}
```

※[7－1]も参考にして下さい。

5-2 複数のキャラを表示する

キャラクターを2つ表示する

複数のキャラクターを表示する場合も、さほど難しくありません。

[5ー1]で使用したD3DXVECTOR3構造体の座標を変更し再度Drawメソッドを呼び出すだけです。

```
    pos.x = 320;
    pos.y = 240;
    pos.z = 0;
    g_pSp->Draw( g_pTex,NULL,NULL,&pos,0xffffffff);

    pos.x = 320;
    pos.y = 240;
    g_pSp->Draw( g_pTex,NULL,NULL,&pos,0xffffffff);
```

多数のキャラクターを表示する

ただ、1つや2つの表示ならともかく、多数のキャラの座標はこの手法ではとても管理できません。

そこで、配列を使い座標をデータ化して管理をしやすくします。

```
static float posdata[3][2] = {
        { 320, 240, },              //0番目のキャラクターの表示座標
        { 160, 360, },              //1番目のキャラクターの表示座標
        { 448, 112, },              //2番目のキャラクターの表示座標
    };

    for( i=0; i<3; i++){
        pos.x = posdata[i][0];
        pos.y = posdata[i][1];
        g_pSp->Draw( g_pTex,NULL,NULL,&pos,0xffffffff);
    }
```

これで、表示座標を変更する際も、修正が容易になります。

5-3 メッセージの表示を管理する

どうやって表示するか

メッセージの表示を管理してみましょう。

メッセージは文字列で構成されていますが、表示をコントロールするためには、文字列を１文字づつ解釈する必要があります。

ここではタスクが１フレームに１度呼ばれる事を利用して、メッセージを解釈、表示します。

メッセージを表示する

例として、通常文字の表示時間の調整と、改行時にボタンを待つ処理を作成していますので、見ていきましょう。

文字の判断

最初に、文字の判断は、switch 文で行なっており、case 文で指定の文字がきたら、それに合わせた処理を行ないます。

通常の文字が処理される場合は、まず指定の時間が過ぎたかどうかをチェックして、過ぎていれば、表示する文字を１文字増やします。

表示する時間

表示の待ち時間は、フレーム単位で DISP_SPEED で定義しています。

改行を処理する時は、ボタンが押されたかをチェックし、押された時だけ、表示する文字を進めます。

ここでは、改行処理だけですが、case 文で指定する文字を増やせば、いろんな文字やコードに対応できますので、改造して色々試してみてください。

LIST 5－3－1 メッセージの表示を管理する

```
void exec05_03(TCB* thisTCB)
{
#define DISP_SPEED 5
typedef struct{
    int             StrPoint;
    int             Time;
} EX05_03_STRUCT ;
```

```
char sample_mess[] =
    "これはメッセージの表示を管理するプログラムの、¥n"
    "テスト用文字列です。¥n"
    "このプログラムを基本に色々変更してみて下さい。";

    EX05_03_STRUCT* work = (EX05_03_STRUCT*)thisTCB->Work;
    RECT font_pos = {   0,   0,640,480,};
    int mess_count;

    switch( sample_mess[work->StrPoint] )
    {
        case  '¥n':  //改行時の処理　Zキーで次行へ
            if( g_DownInputBuff & KEY_Z  )
            {
                work->StrPoint += 1;
                work->Time = 0;
            }
            mess_count = work->StrPoint;
            break;
        case  NULL:   //文字列終了
            mess_count = -1;
            break;

        default: //通常文字の処理
            //一定時間経過後、次の文字へ
            work->Time++;
            if( work->Time == DISP_SPEED )
            {
                work->StrPoint+=2;
                work->Time = 0;
            }
            mess_count = work->StrPoint;
            break;
    }

    g_pFont->DrawText( NULL, sample_mess, mess_count, &font_pos, DT_LEFT,
0xffffffff);

    //Xキーで最初から表示
```

```
    if( g_DownInputBuff & KEY_X  )
    {
        work->StrPoint = 0;
        work->Time     = 0;
    }

}
```

5-4 背景

キャラクターと背景を分ける理由

DirectXにおいて、スクロール等をしない背景の表示処理はキャラクターの表示とほぼ同じです。

それではなぜわざわざ、背景の表示を分けているのでしょうか？

具体的にキャラクターの表示となにが違うのでしょうか？

理由は幾つかありますが、答えとしては「背景の管理」と「プログラムの簡素化」があげられます。

例えば、背景はキャラクターの表示と違い、表示の優先順位はほぼ固定になります（通常は一番奥に表示）。

これは、優先順位が頻繁に入れ替わる他の表示と処理が切り分けられやすい事を意味し、処理が簡略化しやすくなります。

また、背景の切り替えやエフェクト処理、当たり判定などもキャラクター表示とは別扱いにしたほうが楽になるケースが多く、プログラムもすっきりします。

LIST 5－4－1 背景の表示

```
typedef struct{
    int             X;
    int             Y;
} BACK_GROUND;

void BGDraw(BACK_GROUND* bg,int bitmap_id)
{
    D3DXVECTOR3 pos;

    pos.x = bg->X;
    pos.y = bg->Y;
    pos.z = 0;

    g_pSp->Draw( g_pTex[bitmap_id],NULL,NULL,&pos,0xffffffff);
}
```

```
void init05_04(TCB* thisTCB)
{
    //使用するテクスチャの読み込み
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥0022.png",&g_pTex[0] );
}

void exec05_04(TCB* thisTCB)
{
    BACK_GROUND* bg;
    bg = (BACK_GROUND*)thisTCB->Work;

    BGDraw(bg,0);         //背景の描画
}
```

5-5 背景との当たり判定

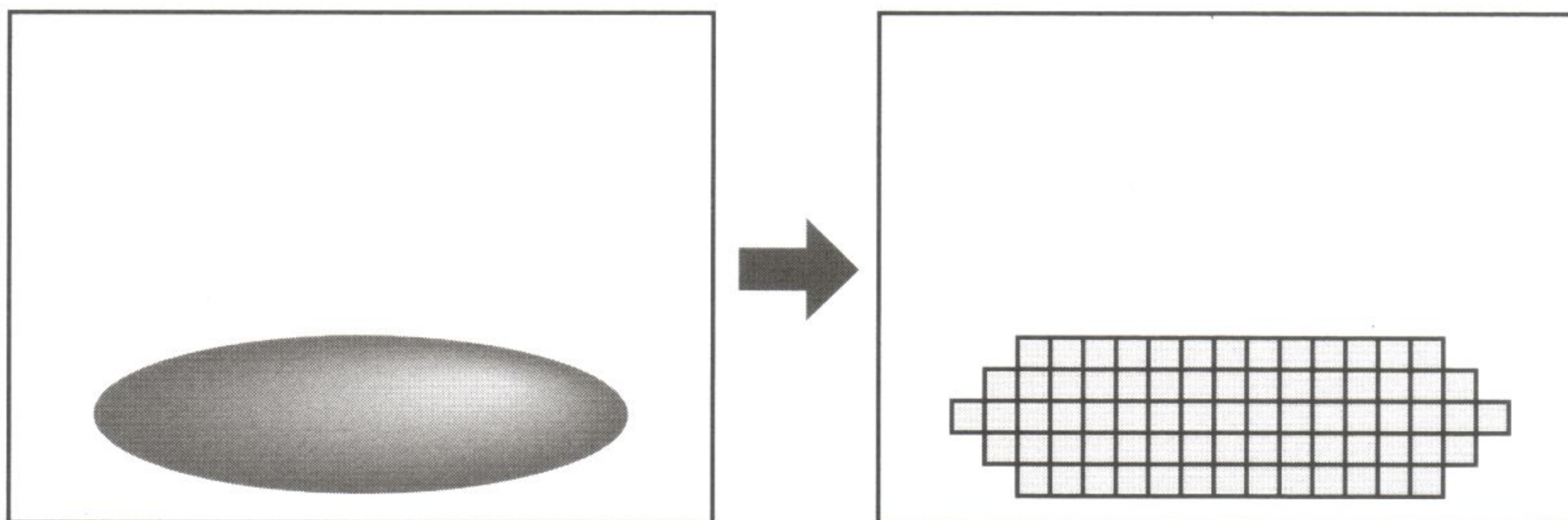
背景との当たり判定を取る方法

表示されている背景に対して当たり判定を取りたくなる事はよくあります。

しかし実際の表示画面に対して、直接当たり判定を取る事は著しく困難です。

このような時は、当たり判定専用の仮想画面を設けて、それに対して処理を行ないます。

図5－5－1 当たり判定仮想画面のイメージ図

実際の画面ではなく、仮想画面を用意して当たり判定を行う

当たり判定のプログラム

仮想画面

ではサンプルプログラムを見ていきましょう。

仮想画面は、2次元配列で用意します。

配列の大きさは画面の大きさや、当たり判定の精度によります。

スクロールをさせるなど画面に対応したり、精度を上げる場合は、使用するデータ容量も上がりますので注意してください。

サンプルでは、640×480の画面、精度を32ドット単位にした場合で20×14の大きさになります。

当たり判定

実際の当たり判定の部分ですが、サンプルではカーソルを動かし、その示す点との当たり判定を取っています。

処理事態はさほど難しくはありません。始めに当たり判定をチェックする座標を仮想画面の精度に合わせます。

合わせた後に、その座標を元に、仮想画面のデータのチェックを行なうと判定を得る事ができます。

ただしこの際、座標の指定方法によっては、配列の範囲外を参照しない様、エラーチェックを行なう必要があります。

エラーチェックを省く事もできますが、その場合は、チェック座標が範囲外を参照しないよう十分注意してください。

LIST 5 ー5ー1 背景との当たり判定

```
typedef struct{
    BACK_GROUND     Bg;
    SPRITE2         Cursor;
} EX05_05_STRUCT ;

void init05_05(TCB* thisTCB)
{
    //使用するテクスチャの読み込み
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥HIT_CHECK.png",&g_pTex[0] );
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥CURSOR.png",&g_pTex[1] );
}

void exec05_05(TCB* thisTCB)
{
#define HIT_ACC 32              //当たり判定の精度

unsigned char check_data[ SCREEN_HEIGHT / HIT_ACC ][ SCREEN_WIDTH / HIT_ACC
] =
{//0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19
{  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, },  //0
{  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, },  //1
{  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, },  //2
```

```
{  0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, },   //3
{  0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, },   //4
{  0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, },   //5
{  0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, },   //6
{  0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, },   //7
{  0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, },   //8
{  0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, },   //9
{  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, },   //10
{  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, },   //11
{  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, },   //12
{  0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, },   //13
{  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, },   //14
};

    EX05_05_STRUCT* work = (EX05_05_STRUCT*)thisTCB->Work;
    BACK_GROUND   bg;
    RECT font_pos = { 0, 0, 640, 480, };
    unsigned int check_x;
    unsigned int check_y;
    BOOL err = FALSE;

    bg.X = 0;
    bg.Y = 0;
    //背景の表示
    BGDraw( &bg, 0 );

    //カーソルの移動
    if( g_InputBuff & KEY_UP    ) work->Cursor.Y -= 8;
    if( g_InputBuff & KEY_DOWN  ) work->Cursor.Y += 8;
    if( g_InputBuff & KEY_RIGHT ) work->Cursor.X += 8;
    if( g_InputBuff & KEY_LEFT  ) work->Cursor.X -= 8;

    SpriteDraw2( &work->Cursor, 1 );

    //背景接触データのチェック
    check_x =  work->Cursor.X / HIT_ACC;
    check_y =  work->Cursor.Y / HIT_ACC;

    //データ範囲外の場合エラー
    if( check_x >= SCREEN_WIDTH  / HIT_ACC) err = TRUE;
```

```
    if( check_y >= SCREEN_HEIGHT / HIT_ACC) err = TRUE;

    //接触判定
    if( !err ) //エラーの場合はチェックしない
    {
        if( check_data[ check_y ][ check_x ] )
        {

            FontPrint(5,15,"HIT BACK GROUND");
        }
    }

}
```

5-6 背景をスクロールさせる

スクロールさせる方法

画面をスクロールさせる方法はいくつかありますが、一番手軽な物は表示画面よりも大きいビットマップを用意しそれを表示する手法です。

スクロールの範囲がそれほど大きくない(2画面分程度)のであれば、この手法で大きな問題はありません。

実際にスクロールをさせるには、スプライトの表示座標と同じように座標を描き換えてやればOKです。

スクロール処理の注意点

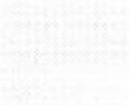
少々間違いやすいのは、スクロールさせる時、スクロール方向は、移動方向とは逆の方向になる事です。

右からスクロールさせたい時は、X座標を一方向へ進めてやらないといけません。

その点さえ間違えなければ、特に難しい所は無いでしょう。

LIST 5－6－1 背景のスクロール

```
void init05_06(TCB* thisTCB)
{
    //使用するテクスチャの読み込み
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥0022.png",&g_pTex[0] );
}

void exec05_06(TCB* thisTCB)
{
#define SCROLL_SPEED 2
    BACK_GROUND* bg;
    bg = (BACK_GROUND*)thisTCB->Work;

    //スクロールは逆方向に進む
    bg->X -= SCROLL_SPEED;
    bg->X %= SCREEN_WIDTH;
    bg->Y = 0;
```

```
    BGDraw(bg,0);      //背景の描画
}
```

5-7 多重スクロール

多重スクロールとは

多重スクロールは遠近感や、立体感を演出するために古くからある手法ですが、その効果は大きく現在でも有効な手法です。

その多重スクロールですが、その手法は大きく分けて３種類あります。

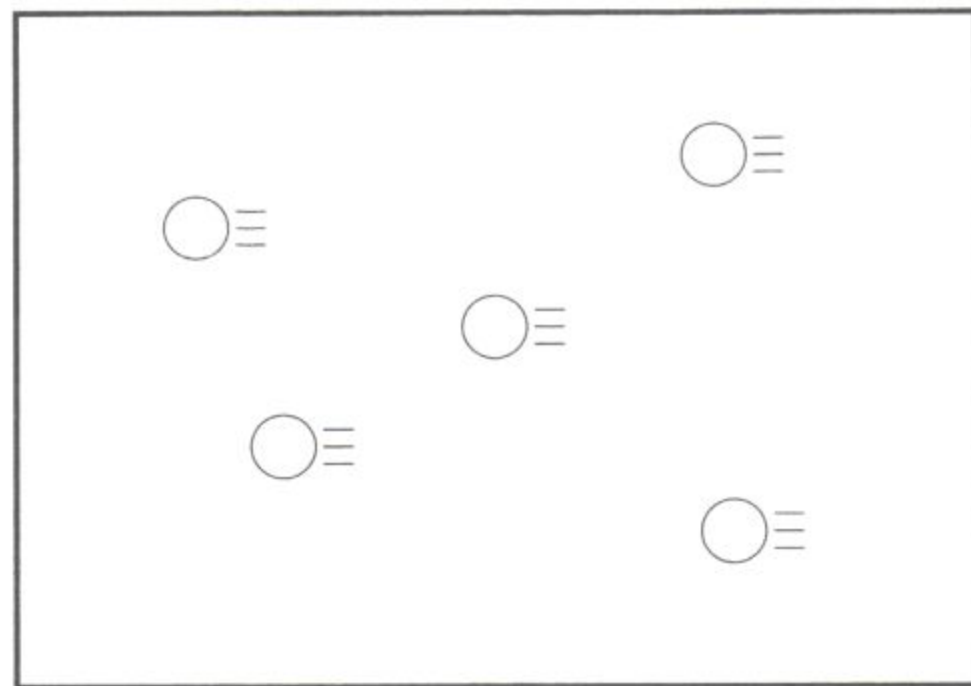
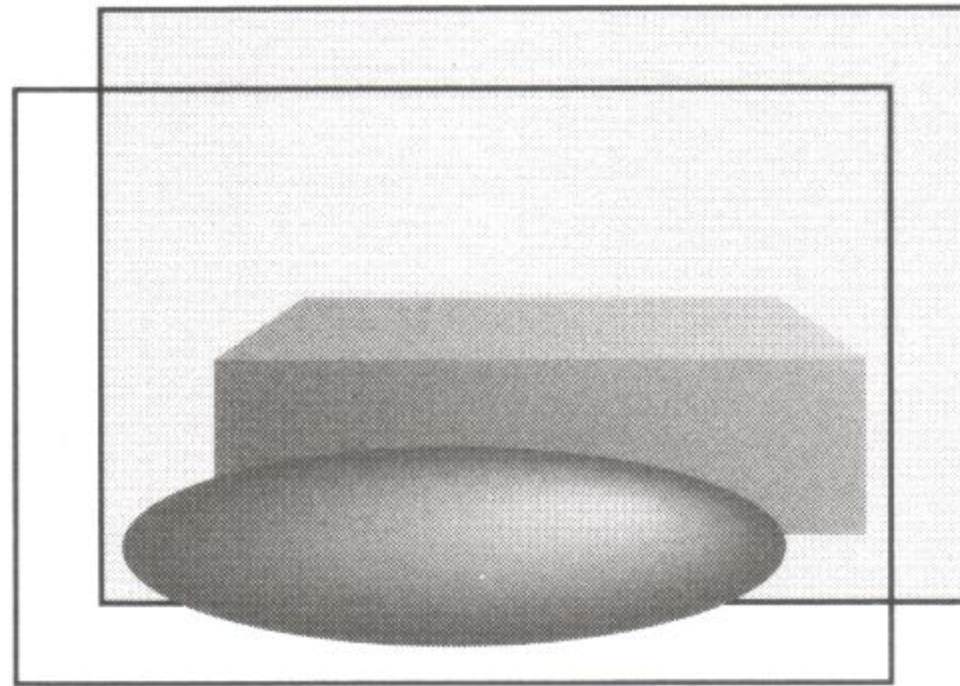
多重スクロールの種類

１つは「スター」や、「パーティクル」などと呼ばれる、小さな表示物を多数表示して、遠近感を出す手法です。

もう１つは背景画面を複数用意し、重ねて表示する手法です。多重スクロールといえばこちらの方が一般的でしょう。

最後の１つは、ラスタースクロールと呼ばれています。こちらの解説は別項にゆずります。

図５ －７－１ スター、背景画面表示のイメージ図

スター
画面上に小さい表示物を多数動かして
スクロールしているように見せる

実際に背景を複数重ねて表示する事で表示

多重スクロールのプログラミング

初期化

では、サンプルリストを見ていきましょう。サンプルでは背景を２つ表示し、同時にスターの処理を行なっています。

まず、初期化です。使用するテクスチャを読み込んだ後に、表示するスターの座標を初期化しています。

スターの座標は乱数にしていますが、数は32個で固定です。

スターのスクロール速度も、個別で変えたほうが遠近感が出るため、乱数で設定します。

最後に背景の座標を初期して、初期化処理は終了です。

```
//スター座標の初期化
for( loop = 0; loop < 32; loop++)
{
    work->StarPosX[ loop ]  = rand() / (RAND_MAX / SCREEN_WIDTH);
    work->StarPosY[ loop ]  = rand() / (RAND_MAX / SCREEN_HEIGHT);
    work->StarSpeed[ loop ] = rand() / (RAND_MAX / 2.0) + 1.0;
}
//手前表示背景のY座標
work->Bg1.Y  = 416;
```

多重スクロールの処理

次に実際の処理部分です。

まずは、一番奥の背景を表示します。これは、単純に読み込んだ背景を表示してやります。

次にスター部分の処理です。スターは乱数で設定した速度でスクロールさせます。

もし、スターの表示座標が画面左端に達したら、表示座標を右端に戻してやります。

これで、スターは終わりなくスクロールする事ができます。これをスターの個数分繰り返します。

最後に、手前の背景の表示です。

一番最後に表示処理をした物が、一番手前に表示されるため、手前に表示する背景も一番最後に処理してやります。

もし、スター画面を背景の手前に表示するのであれば、スター処理を一番最後にするとよいでしょう。

LIST 5－7－1 多重スクロール

```
#define STAR_MAX 32
typedef struct{
    BACK_GROUND    Bg0;
    BACK_GROUND    Bg1;
    float          StarPosX[STAR_MAX];
    float          StarPosY[STAR_MAX];
```

```
    float           StarSpeed[STAR_MAX];
    RECT*           StarPix;
} EX05_07_STRUCT ;

void init05_07(TCB* thisTCB)
{
    EX05_07_STRUCT* work = (EX05_07_STRUCT*)thisTCB->Work;
    int loop;

    //使用するテクスチャの読み込み
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥SPACE_BG.png",&g_pTex[0] );
    D3DXCreateTextureFromFile( g_pD3Ddevice, "..¥¥..¥¥data¥¥pix.png",
&g_pTex[1] );
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥GRAND_BG.png",&g_pTex[2] );

    //スター座標の初期化
    for( loop = 0; loop < STAR_MAX; loop++)
    {
        work->StarPosX[ loop ]  = rand() / (RAND_MAX / SCREEN_WIDTH);
        work->StarPosY[ loop ]  = rand() / (RAND_MAX /
SCREEN_HEIGHT);
        work->StarSpeed[ loop ] = rand() / (RAND_MAX / 2.0) + 1.0;
    }
    //手前表示背景のY座標
    work->Bg1.Y  = 416;
}

void exec05_07(TCB* thisTCB)
{
    EX05_07_STRUCT* work = (EX05_07_STRUCT*)thisTCB->Work;
    SPRITE sprt;
    RECT star_rect = { 0, 0, 2, 2 };
    int loop;

    //一番奥の背景
    BGDraw( &work->Bg0, 0 );

    //スター描画
```

```
    sprt.SrcRect = &star_rect;

    for( loop = 0; loop < STAR_MAX; loop++)
    {
        work->StarPosX[ loop ]  -= work->StarSpeed[ loop ];
        //左端に到達したら右端に戻す
        if( work->StarPosX[ loop ]  < 0 )
            work->StarPosX[ loop ]  = SCREEN_WIDTH;

        sprt.X = work->StarPosX[ loop ];
        sprt.Y = work->StarPosY[ loop ];

        SpriteDraw( &sprt, 1 );
    }
    //一番手前の背景
    work->Bg1.X  -= 4.0;
    //左端に到達したら右端に戻す
    if( work->Bg1.X < -(SCREEN_WIDTH / 2) ) work->Bg1.X = 0;

    BGDraw( &work->Bg1, 2 );}
}
```

5-8 拡大、縮小

DirectXでの拡大、縮小

スプライトの拡大縮小は初心者には若干敷居が高いようです。

以前は拡大縮小率を指定してやるだけで比較的手軽に実現できたのですが、現在のDirectXのバージョンでは仕様が変わってしまい、拡大縮小に変換行列を指定しなくてはならなくなりました。

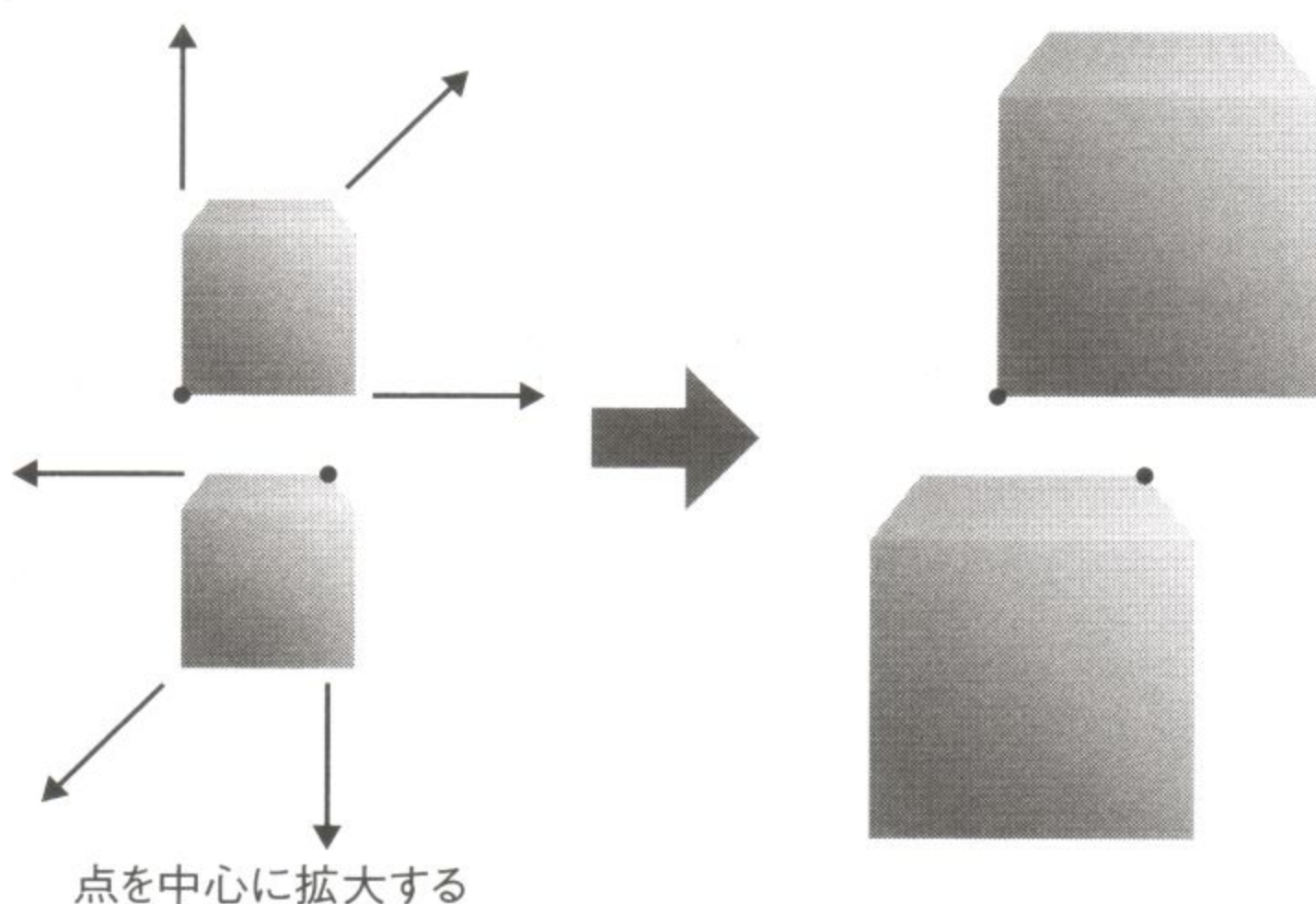
行列と聞いて、頭を痛める人もいるかもしれませんが、手順はほぼ固定なので、そう心配する事はありません。

拡大、縮小のプログラム

座標の設定

ここでは、まず拡大縮小に必要な、座標を設定しています。設定後、実際の拡大縮小処理はex5_8_CustomDrawで行ないます。

変数Centerは、拡大縮小を行なう際の中心点です。たまに勘違いをする人がいるのですが、拡大縮小にも中心点が必要です。

図5－8－1 拡大縮小中心のイメージ図

点を中心に拡大する

なお、ここでは、拡大縮小のアニメーションをするのに、sin関数を使用しています。sin関数は周期関数※ですので、こういった数値の繰り返しを行なう処理には便利に使う事が出来ます。

※周期関数（一定周期で値が繰り返される関数）。

LIST 5 －8－1 拡大、縮小

```
typedef struct{
    float           X;
    float           Y;
    D3DXVECTOR3     Center;
    D3DXVECTOR3     Scale;
    float           Count;
} EX5_8_STRUCT ;

void init05_08(TCB* thisTCB)
{
    //使用するテクスチャの読み込み
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥0011.png",&g_pTex[0] );
}

void exec05_08(TCB* thisTCB)
{
    EX5_8_STRUCT* work = (EX5_8_STRUCT*)thisTCB->Work;

    work->X = 320 - 32;
    work->Y = 240 - 32;
    work->Center.x = 32;
    work->Center.y = 32;

    work->Count += 0.125;
    work->Scale.x = 1.0 + sin( work->Count ) * 0.5;
    work->Scale.y = 1.0 + sin( work->Count ) * 0.5;


    ex5_8_CustomDraw(work,0);
}
```

次に実際の描画関数です。基本的には、通常の描画関数に似ています。

ここでのポイントは、D3DXMatrixScalingです。ここで、スケール値を拡大縮小の変換行列に変換しています。

その後、SetTransform関数でDirectX側に変換行列を設定します。

最後にDraw関数で描画すれば終了です。

LIST 5 −8−2 描画関数

```
void ex5_8_CustomDraw( EX5_8_STRUCT* work, int bitmap_id )
{
    D3DXVECTOR3 pos;
    D3DXMATRIX  matrix;


    pos.x = work->X / work->Scale.x;
    pos.y = work->Y / work->Scale.y;
    pos.z = 0;

    D3DXMatrixScaling( &matrix, work->Scale.x, work->Scale.y, 1.0 );
    g_pSp->SetTransform( &matrix );
    g_pSp->Draw( g_pTex[0], NULL, &work->Center, &pos, 0xffffffff);
}
```

5-9 回転

DirectXの回転

回転は見た目も派手で、ゲームの画面効果でも頻繁に使われる処理です。

ただ、DirectXでは3D関連との整合性を取るためと思われますが、回転の指定方法に変換行列を使うようになっています。

そのため、思い通りの座標で回転をしようとするための手順が、かなり複雑になってしまっています。

回転のプログラム

座標の設定

ここでの処理は比較的簡単です。まず表示位置と、回転座標の中心点を設定します。

その後に、毎フレームごとに回転度数を加算していき、常にビットマップが回転するようになっています。

LIST 5－9－1 回転

```
typedef struct{
    float           X;
    float           Y;
    D3DXVECTOR3     Center;
    float           Rotate;
} EX5_9_STRUCT ;

void init05_09(TCB* thisTCB)
{
    //使用するテクスチャの読み込み
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥¥..¥¥data¥¥0011.png",&g_pTex[0] );
}

void exec05_09(TCB* thisTCB)
{
    EX5_9_STRUCT* work = (EX5_9_STRUCT*)thisTCB->Work;
```

```
    work->X = 320 ;
    work->Y = 240 ;
    work->Center.x = 320 + 64;
    work->Center.y = 240 + 64;

    work->Rotate += 0.125;

    ex5_9_CustomDraw(work,0);
}
```


回転行列を取得

ここからが、少々厄介です。

まず、D3DXMatrixRotatinZを使い、Z軸を中心とした、回転行列を取得します。

詳しい説明は長くなるので省略しますが、ここでは「2Dでの回転は3DのZ軸を中心とした回転行列として扱う」ものと、とらえてください。

平行移動行列を取得

次にD3DMatrixTranslationで平行移動行列を取得します。

平行移動行列とは平たく言うとX、Y座標を行列にした物です。

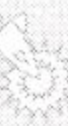
取得した行列を合成

最後にD3DXMatrixMultiplyで取得した回転行列と平行移動行列を、1つの行列に合成します。

後は、SetTransform関数で変換行列を設定し、Draw関数で表示します。

LIST 5 −9−2

```
void ex5_9_CustomDraw( EX5_9_STRUCT* work, int bitmap_id )
{
    D3DXVECTOR3 pos;
    D3DXMATRIX  matrix_a;
    D3DXMATRIX  matrix_b;
    D3DXMATRIX  matrix;

    pos.x = work->X;
    pos.y = work->Y;
    pos.z = 0;

    D3DXMatrixRotationZ( &matrix_a, work->Rotate );                //回転行列を取得
```

```
    D3DXMatrixTranslation( &matrix_b, pos.x, pos.y, 0 );          //平行移動行列を取得

    D3DXMatrixMultiply( &matrix, &matrix_a, &matrix_b );          //行列を合成

    g_pSp->SetTransform( &matrix );


    g_pSp->Draw( g_pTex[0], NULL, &work->Center, &pos, 0xffffffff);
}
```

5-10 反転処理

DirectX のスプライト反転

DirectX で反転を行なってみましょう。

反転は古くからある画像処理で、ファミコンにもその機能がハードウェアで搭載されています。

ただ、DirectX のスプライトで反転を行なうには、拡大縮小と同様に行列を使う必要があり、少々ややこしくなっています。

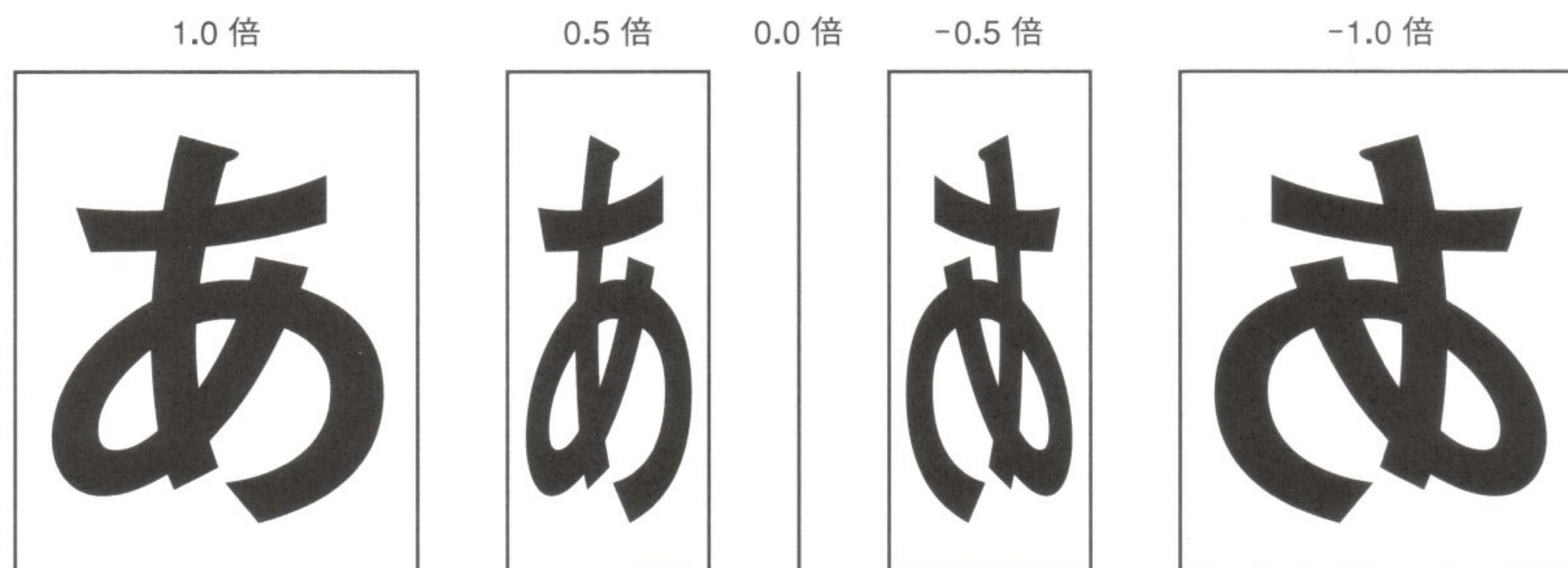
反転処理のプログラム

サンプルプログラムですが、反転機能を扱う専用の描画関数を作成する事で、反転処理を行なっています。

反転処理そのものは、反転のための行列を用意して行なうのですが、ここではもっと手軽に、負方向へのスケール値を設定する事で実現しています。

図5 －10－1

1.0 倍　0.5 倍　0.0 倍　-0.5 倍　-1.0 倍

あ　あ　あ　あ

では、実際のプログラムを見てみましょう。

初期化

まず、最初に拡大縮小のスケール値と中心点を初期化しています。

スケール値の設定

次に反転表示を行なうためのフラグをチェックして、スケール値を設定する処理を行ないます。

反転には、X方向の反転と、Y方向の反転の2種類があるため、それぞれの方向について処理を行ないます。

座標と行列を作成／描画

反転処理を行なったら、その値を元に、座標の設定と変換行列の作成、設定を行ないます。

最後に、描画をして終了です。

LIST 5 －10－1 反転処理

```
typedef struct{
    float           X;
    float           Y;
    D3DXVECTOR3     Center;
    D3DXVECTOR3     Scale;
    BOOL            ReversX;
    BOOL            ReversY;
} EX5_10_STRUCT ;

void ex5_10_CustomDraw( EX5_10_STRUCT* work, int bitmap_id )
{
    D3DXVECTOR3 pos;
    D3DXMATRIX  matrix;

    work->Scale.x = 1.0;
    work->Scale.y = 1.0;
    work->Center.x = 0;
    work->Center.y = 0;

    if( work->ReversX )  //X反転処理
    {
        work->Scale.x = -1.0;
        work->Center.x = 128;
    }

```

```
		if( work->ReversY ) //Y反転処理
		{
			work->Scale.y = -1.0;
			work->Center.y = 128;
		}
		//座標設定
		pos.x = work->X / work->Scale.x;
		pos.y = work->Y / work->Scale.y;
		pos.z = 0;

		//変換行列の設定
		D3DXMatrixScaling( &matrix, work->Scale.x, work->Scale.y, 1.0 );
		g_pSp->SetTransform( &matrix );
		//描画
		g_pSp->Draw( g_pTex[0], NULL, &work->Center, &pos, 0xffffffff);
}

void init05_10(TCB* thisTCB)
{
		//使用するテクスチャの読み込み
		D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥0011.png",&g_pTex[0] );
}

void exec05_10(TCB* thisTCB)
{
		EX5_10_STRUCT* work = (EX5_10_STRUCT*)thisTCB->Work;


		work->X = 64;
		work->Y = 64;
		work->ReversX = 0;
		work->ReversY = 0;
		ex5_10_CustomDraw(work,0);

		work->X = 384;
		work->Y = 64;
		work->ReversX = 1;
		work->ReversY = 0;
		ex5_10_CustomDraw(work,0);
```

```
	work->X = 64;
	work->Y = 288;
	work->ReversX = 0;
	work->ReversY = 1;
	ex5_10_CustomDraw(work,0);

	work->X = 384;
	work->Y = 288;
	work->ReversX = 1;
	work->ReversY = 1;
	ex5_10_CustomDraw(work,0);
}
```

5-11 αブレンド

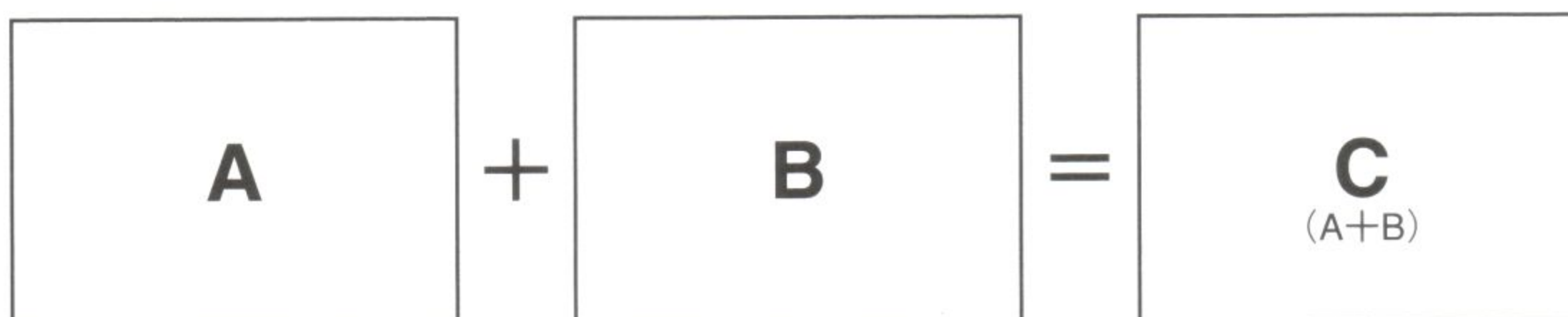
半透明

αブレンドとは平たく言うと、半透明機能の事です。

半透明は、画像を描画する際に、描画する画像と描画される画像との演算を行なう事で実現します。

図5－11－1 画像間の演算のイメージ図

画像A と画像B の間で指定した演算が行なわれ画像C になる

A	＋	B	＝	C (A＋B)

各画像ではドット単位で演算が行なわれている例
加算による合成の場合

R	G	B		R	G	B		R	G	B
255	0	0	＋	0	0	255	＝	255	0	255
(画像A) 赤				(画像B) 青				(画像C) 紫		

ただ、一口に演算といっても沢山の種類があり、またその演算種類を、描画する画像と描画される画像、それぞれに設定しなくてはなりません。

ただし、一度設定してしまえば、演算の種類を変更しない限り、再度設定する必要はありません。

DirectXでの設定

アルファブレンドの設定

```
g_pD3Ddevice->SetRenderState( D3DRS_SRCBLEND,  D3DBLEND_SRCALPHA );
g_pD3Ddevice->SetRenderState( D3DRS_DESTBLEND, D3DBLEND_INVSRCALPHA );
g_pD3Ddevice->SetRenderState( D3DRS_ALPHABLENDENABLE, TRUE );
```

上記のリストは、DirectXの初期化処理の一部です。

演算種類の指定は、DirctXの表示デバイスに対して行ない、メソッドSetRenderStateで設定します。

なお、半透明機能を使用する際はSetRenderStateを使って、αブレンドの機能をオンにしなくてはいけません。

描画

一度設定してしまえば、後は描画するだけです。

描画の際の第5引数の、最上位8ビットに半透明の度合い（α値）を設定すれば半透明で描画されます。

α値は0xffで不透明、数値が下がると徐々に透明になって行き、0x00で完全に透明になります。

```
g_pSp->Draw( g_pTex[0], NULL, NULL, &pos, 0x80ffffff);
```

この部分がα値

5-12 ウィンドウの表示

ゲーム中のウィンドウ

もちろんウィンドウといっても、OSの機能を使った物ではなく、ゲーム中で使用されるウィンドウの事です。

RPG等も含めゲームではテキスト関連の表示はウィンドウで表示される事が多く、デザインも非常にこった物が多くなっています。

「上書き型」の処理

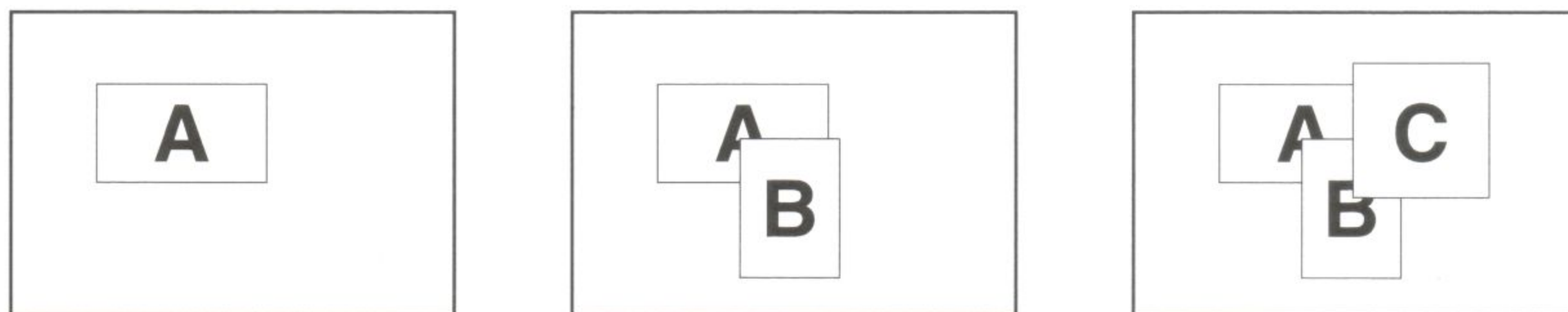
ではそれらのウィンドウは一体どうやって処理されているのでしょうか？

ウィンドウの描画には幾つか種類がありますが、ここで紹介するのは、「上書き型」と呼ばれるものです。

ウィンドウは、「枠」と「表示テキスト」から構成され、一番奥から、表示の優先順位に合わせて描画していきます。

後に描かれた方が、以前に描かれた「ウィンドウ」を上書きするため、「上書き型」と呼ばれます。

図5－12－1 上書き型ウィンドウのイメージ図

次々と矩形を上書きする事によって、ウィンドウ処理を行なう

この方式は非常に単純で分かりやすいのですが、多数の大きなウィンドウを作成すると、それに比例して表示の処理がとても重くなってしまいます。

ですが、ゲームでは多数のウィンドウを使用することは余りないので、ゲームに向いているといえるでしょう。

ウィンドウの表示

では、実際のプログラムを解説していきます。

サンプルでは、1ウィンドウ＝1タスクで構成されており、1つのタスクで、ウィンドウ(表示文字と枠)を管理しています。

初期化

まず初期化ですが、使用するテクスチャを読み込んだ後、タスクを2つ作成しています。

説明したように、1タスク＝1ウィンドウとしていますので、メインの処理も含めて、プログラムでは3つのウィンドウを描画しています。

表示の優先順位を変えるには

ウィンドウの表示優先は、タスクの優先順位となっているため、ウィンドウ表示の優先順位を変える時は、タスクの優先順位を変える事で行ないます。

ウィンドウの描画

肝心のウィンドウ処理部分ですが、非常にシンプルなものになっています。

まず、ウィンドウの大きさを定義し、そのデータを元に、枠を表示する関数を呼び出し、枠を描画します。

その後、枠の上にテキストを表示してやります。

枠の処理

ウィンドウの処理は以上となりますが、最後に枠の処理についてふれておきます。

枠の描画は、関数exec5_12_WinDrawで行なっており、引数に枠の大きさを格納したRECT構造体と、枠のタイプを指定するIDを渡す様になっています。

処理内容は単純で、指定された位置と大きさの矩形を描画するだけですが、あえてスプライトを用い、ドットを拡大して矩形描画しています。

固有の枠を使用する場合は、この関数を改造して単純なスプライト表示にする等、改良してみると良いでしょう。

LIST 5 －12－1 ウィンドウを表示する

```
void exec5_12_WinDraw( RECT* pRect, int ID )
{
    D3DXVECTOR3 pos;
    D3DXMATRIX  matrix;
    RECT win_rect[] =
```

```
{
    { 0, 0, 1, 1,},
    { 7, 1, 8, 2,},
    {10,10,11,11,},
};
    float scaleWidth;
    float scaleHeight;


    //ウィンドウ幅と高さ設定
    scaleWidth  = pRect->right  - pRect->left;
    scaleHeight = pRect->bottom - pRect->top;

    //表示座標設定
    pos.x = pRect->left / scaleWidth;
    pos.y = pRect->top  / scaleHeight;
    pos.z = 0;

    D3DXMatrixScaling( &matrix, scaleWidth, scaleHeight, 1.0 );
    g_pSp->SetTransform( &matrix );
    g_pSp->Draw( g_pTex[0], &win_rect[ ID ], NULL, &pos, 0xffffffff);

    //設定した行列を元に戻す
    D3DXMatrixScaling( &matrix, 1.0, 1.0, 1.0 );
    g_pSp->SetTransform( &matrix );

}

void exec05_12_window3(TCB* thisTCB)
{
   RECT win_rect = { 128, 256, 512, 320 };
   char str[128];

    //ウィンドウ枠表示
    exec5_12_WinDraw( &win_rect, 1 );

    //ウィンドウ内部の文字表示
    FontPrint( win_rect.left , win_rect.top, "SAMPLE¥nWINDOW2¥nCOLOR
GRADATION¥nTEST!!");
}
```

```
void exec05_12_window2(TCB* thisTCB)
{
    RECT win_rect = { 336, 264, 496, 416 };
    char str[128];
     //ウィンドウ枠表示
     exec5_12_WinDraw( &win_rect, 0 );

     //ウィンドウ内部の文字表示
     FontPrint( win_rect.left , win_rect.top, "¥n 1.MENU¥n 2.STATUS¥n
3.ITEM¥n 4.HELP¥n 5.OPTION¥n 6.ETC¥n 7.");
}

void init05_12(TCB* thisTCB)
{
     //使用するテクスチャの読み込み
     D3DXCreateTextureFromFile( g_pD3Ddevice, "..¥¥..¥¥data¥¥pix.png",
&g_pTex[0] );
     //2つ目のウィンドウ
       TaskMake(exec05_12_window2,0x1000);
     //3つ目のウィンドウ
       TaskMake(exec05_12_window3,0x1000);
}

void exec05_12(TCB* thisTCB)
{
    RECT win_rect = { 64, 384, 576, 448 };
    char str[128];

     //ウィンドウ枠表示
     exec5_12_WinDraw( &win_rect, 2 );

     //ウィンドウ内部の文字表示
     FontPrint( win_rect.left , win_rect.top, "¥n SAMPLE WINDOW MAIN¥n MAKE
FIRST TASK");
}
```

5-13 広いマップの表示

マップチップ

［5－7］で使用した背景の表示手法だと、手軽ではあるのですが、ある程度以上の大きさの画像を表示するとなると、非常に巨大なメモリを消費してしまいます。

例えば、RPGなどで縦横それぞれ32画面分の大きさの背景をスクロール表示させる事を考えて見ましょう。

すると、画像のデータだけで640×480×32×32となり、256色の画像だとしても300MByteを超えてしまいます。

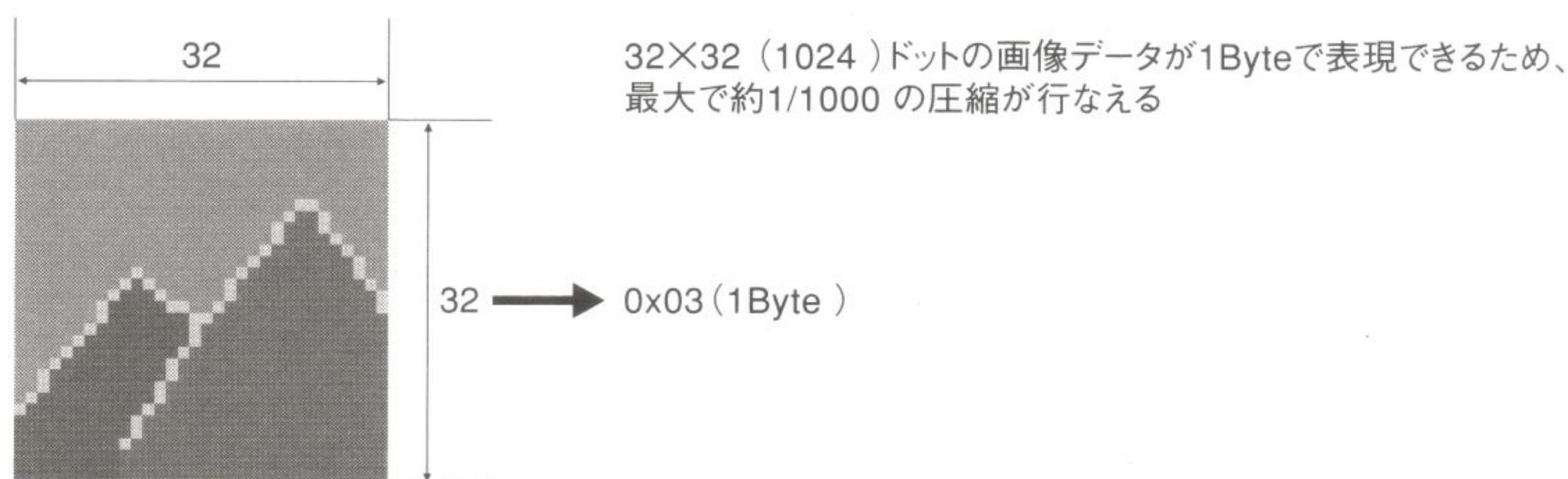
昨今のマシンが大きなメモリを持っているとはいえ、これでは多重スクロールやアニメーションもままなりません。

そこで、一般的には、マップチップまたはタイルと呼ばれる手法を使って、容量を節約します。

図5－13－1 容量節約のイメージ図

32

32×32（1024）ドットの画像データが1Byteで表現できるため、
最大で約1/1000の圧縮が行なえる

32 → 0x03（1Byte）

マップチップとは平たくいうと、固定サイズの圧縮手法※の一種です。

画像は「チップ」または「タイル」と呼ばれる固定されたサイズの小さなグラフィックのみを用いて構成されます。

チップには個別に番号が振られ、この番号でデータを管理する事により、容量を節約します。

※一般で言う圧縮とは違い、どちらかというとパレット画像のイメージに近い。

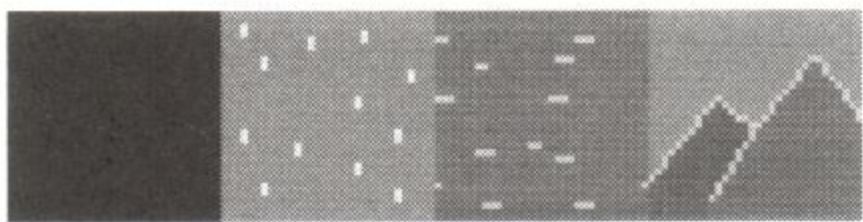
どれだけ節約できる？

どれだけ節約できるかはこの番号の数（チップの種類）と、チップのサイズによって変わりますが、32×32のチップ256種類(1Byte)で画像を構成した場合、その節約率は約1000分の1です。

上記の例に当てはめますと、300MByteのデータが、0.3MByteになり、大幅な容量節約になります。

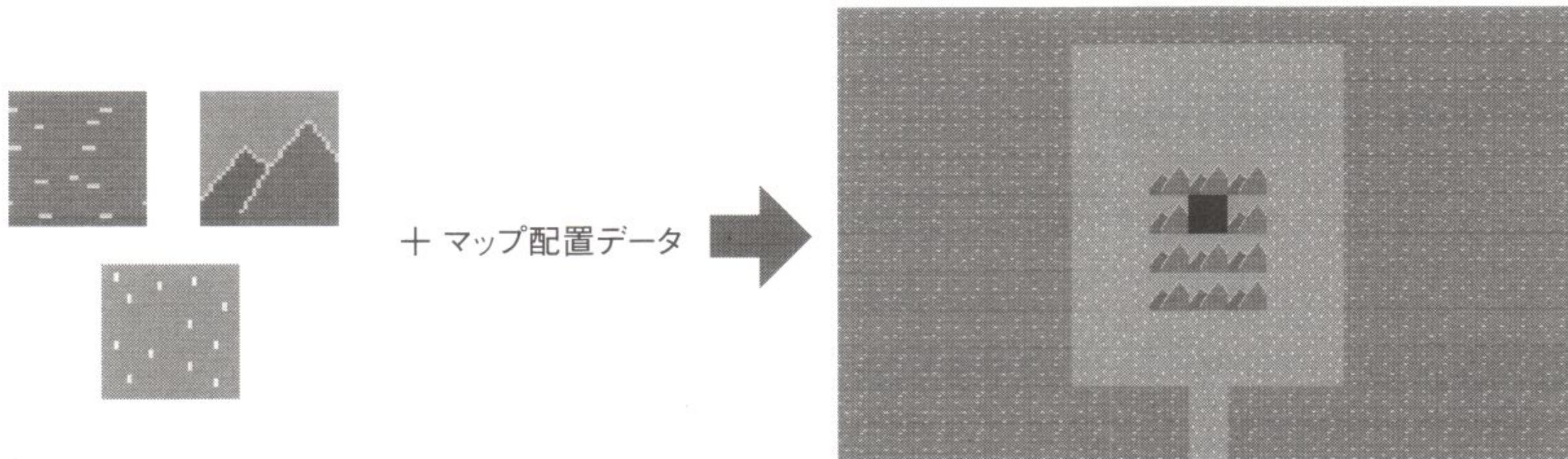
チップを表示する方法

では実際にどの様に作っていくかを見てみましょう。

図5－13－2 実際のチップデータ図

これはサンプルで使われているチップのデータです。32×32の画像を4つ用意しています。このチップをマップデータと呼ばれるチップの並びを記述した、データの通りに表示します。

表示は対応する画面座標のチップを、マップデータから取得し、そのチップを描画します。対応する画面座標の計算はX、Yそれぞれに対する単純な2重ループです。

図5－13－3 チップデータの表示座標引用のイメージ図

＋ マップ配置データ

チップデータ自体はバラバラな為、データに沿って1枚ずつ表示する処理が必要

座標の取得方法

取得するマップデータの座標は、実際のスクロール座標を、チップのサイズで割る事で取得できます。

```
	//チップデータのX座標を計算
	map_dataX =  work->ScrollX / CHIP_SIZE + loopX;
	if( map_dataX < 0) map_dataX += MAP_SIZE_X * ( -map_dataX / MAP_SIZE_X + 
1 );
	map_dataX %= MAP_SIZE_X;
```


ただ、そのまま割っただけでは、マップのサイズを越えるデータを取得してしまう場合があります。

そこでデータ取得の座標を補正する必要があります。

特に負の値の場合座標がずれてしまうので、注意が必要です。ここでは負の座標の大きさの分だけデータ取得座標を加算する事(正座標に補正)で対処しています。

最後に、表示座標にチップを表示してやります。表示座標はloopの値をチップサイズに掛ける事で計算できます。

ドット単位でスクロール

ただ、そのままloopの値を使ってしまうとチップサイズの大きさでしかスクロールしませんので注意してください。

ドット単位でスクロールさせるには、実際のスクロール座標の端数値を引いてやる必要があります。

```
	//チップの表示座標を計算
	chip.X = loopX * CHIP_SIZE - work->ScrollX % CHIP_SIZE - CHIP_SIZE;
```

LIST 5 －13－1 広いマップを表示する 全リスト

```
typedef struct{
	int		ScrollX;
	int		ScrollY;
} EX05_13_STRUCT ;

void init05_13(TCB* thisTCB)
{
	//使用するテクスチャの読み込み
```

```
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥MAP_CHIP.png",&g_pTex[0] );
}

void exec05_13(TCB* thisTCB)
{
//マップチップの大きさ
#define CHIP_SIZE 32
//マップのサイズX
#define MAP_SIZE_X 40
//マップのサイズY
#define MAP_SIZE_Y 15

unsigned char map_data[ MAP_SIZE_Y ][ MAP_SIZE_X ] =
{//0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
25 26 27 28 29 30 31 32 33 34 35 36 37 38 39
{  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2,
2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, },  //0
{  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2,
2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, },  //1
{  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2,
2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, },  //2
{  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2,
2, 2, 2, 2, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, },  //3
{  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2,
2, 2, 2, 2, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, },  //4
{  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2,
2, 2, 2, 2, 1, 3, 3, 3, 1, 2, 2, 2, 2, 2, 2, 2, },  //5
{  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2,
2, 2, 2, 2, 1, 3, 0, 3, 1, 2, 2, 2, 2, 2, 2, 2, },  //6
{  2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 2, 2, 2, 2,
2, 2, 2, 2, 1, 3, 3, 3, 1, 2, 2, 2, 2, 2, 2, 2, },  //7
{  2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 2, 2, 2, 2,
2, 2, 2, 2, 1, 3, 3, 3, 1, 2, 2, 2, 2, 2, 2, 2, },  //8
{  2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 2, 2, 2, 2,
2, 2, 2, 2, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, },  //9
{  2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 2, 2, 2, 2,
2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, },  //10
{  2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 2, 2, 2, 2,
2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, },  //11
{  2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 2, 2, 2, 2,
```

```
2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, },  //12
{  2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 2, 2, 2, 2,
2, 2, 2, 2, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, },  //13
{  2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 2, 2, 2, 2,
2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, },  //14
};

    EX05_13_STRUCT* work = (EX05_13_STRUCT*)thisTCB->Work;
    int loopX;
    int loopY;
    //取得するマップデータのX座標
    int map_dataX;
    //取得するマップデータのY座標
    int map_dataY;

    SPRITE2  chip;
    RECT chip_ptn[4] = {
    {  0,   0,  32, 32},
    { 32,   0,  64, 32},
    { 64,   0,  96, 32},
    { 96,   0, 128, 32},
    };

    //スクロール座標の計算処理
    if( g_InputBuff & KEY_UP    ) work->ScrollY -= 8;
    if( g_InputBuff & KEY_DOWN  ) work->ScrollY += 8;
    if( g_InputBuff & KEY_RIGHT ) work->ScrollX += 8;
    if( g_InputBuff & KEY_LEFT  ) work->ScrollX -= 8;

    for( loopY = 0; loopY < SCREEN_HEIGHT / CHIP_SIZE + 2; loopY++ )
    {
        for( loopX = 0; loopX < SCREEN_WIDTH / CHIP_SIZE + 2; loopX++ )
        {
            //チップデータのX座標を計算
            map_dataX =  work->ScrollX / CHIP_SIZE + loopX;
            if( map_dataX < 0)
                map_dataX += MAP_SIZE_X * ( -map_dataX / MAP_SIZE_X + 1 );
            map_dataX %= MAP_SIZE_X;

            //チップデータのY座標を計算
```

```
            map_dataY =  work->ScrollY / CHIP_SIZE + loopY;
            map_dataY %= MAP_SIZE_Y;
            if( map_dataY < 0)
                map_dataY += MAP_SIZE_Y * ( -map_dataY / MAP_SIZE_Y + 1 );

            //チップの表示座標を計算
            chip.X = loopX * CHIP_SIZE - work->ScrollX % CHIP_SIZE -
CHIP_SIZE;
            chip.Y = loopY * CHIP_SIZE - work->ScrollY % CHIP_SIZE -
CHIP_SIZE;

            //描画するチップのデータを取得
            chip.SrcRect = &chip_ptn[  map_data[ map_dataY ][ map_dataX ]
];

            SpriteDraw( &chip, 0);
        }
    }
}
```

5-14 背景のアニメーション

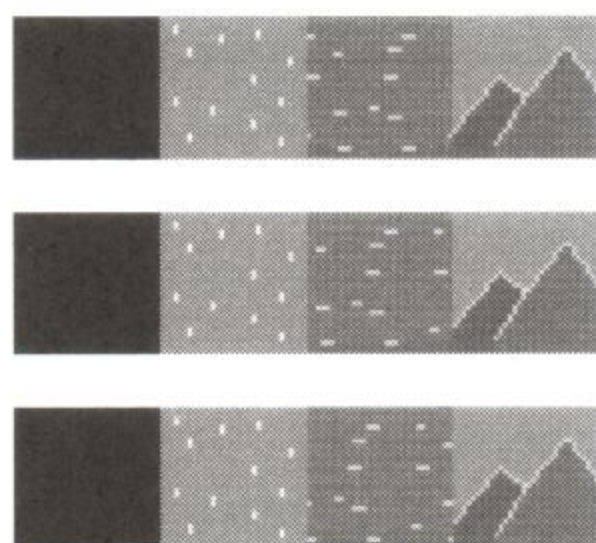
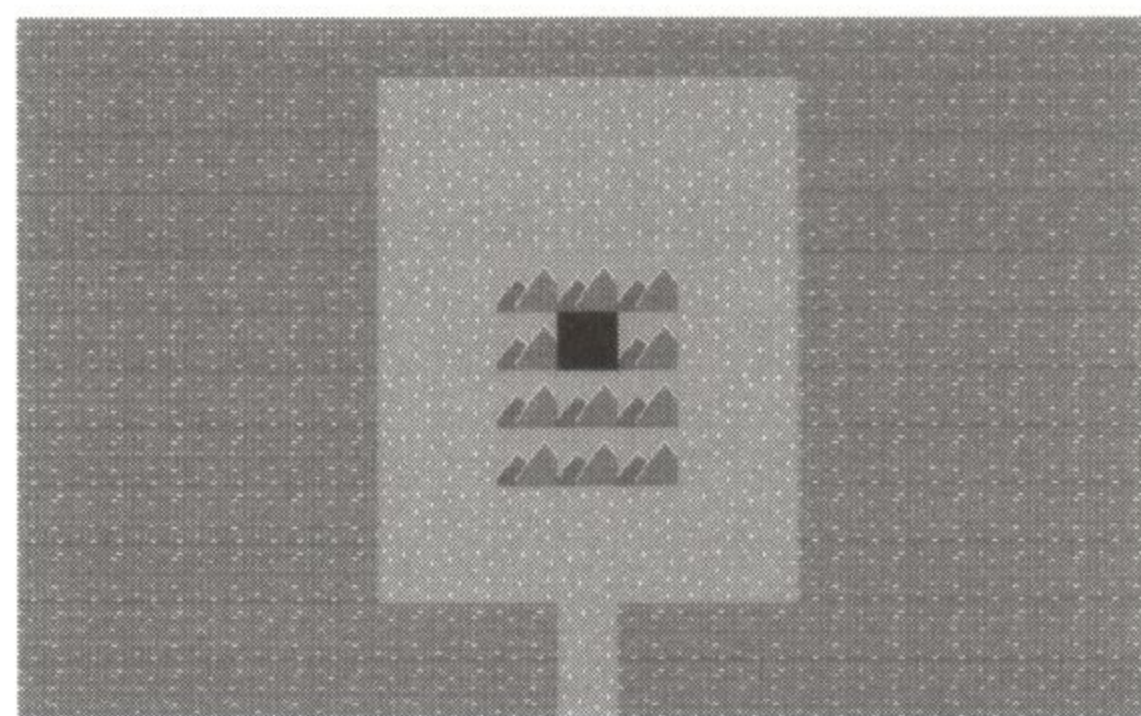
チップを利用して背景アニメーション

背景のアニメーションについて考えてみます。

通常、アニメーション処理は、描画される絵を切り変える事によって行なわれます。

ただし、背景の場合、その切り替えるデータが膨大になってしまうため、通常はマップチップで構成された背景に対してのみアニメーションを行ないます。

こうする事により、画面全体のアニメーションに比べ、変更箇所がマップチップのみになるため、大幅にアニメーション容量を節約する事ができます。

図5－14－1 マップチップの使える背景のみアニメーションを適用するイメージ図

背景アニメーションのプログラム

データの読み込み

ではサンプルプログラムを見ていきましょう。

まず、初期化時にアニメーションデータを読み込みます。

サンプルでは４パターンのアニメーションを行なうため、それに合わせて４種類のデータを読み込みます。

カウンタの計算

初期化後、アニメーションを行なうためのカウンタを計算します。

変数AnimCountにアニメーションの速度を毎フレーム加算してやります。

この時、もしカウンタが、アニメーションのパターン数を超えたら、補正してやる必要があります。

補正処理はカウンタから、アニメーションのパターン数を引いてやるだけです。

表示の処理

最後に、マップチップの表示です。

こちらは、表示用のパターンに合わせて、描画するスプライトを切り替えてやるだけです。

なお、マップチップの表示処理自体は、[5－13]と同じですが、スクロール処理を行なわないため、ずっとシンプルになっています。

固定画面でマップチップを使用する場合は、こちらを参考にしても良いでしょう。

LIST 5－14－1 背景のアニメーション

```
typedef struct{
    int             ScrollX;
    int             ScrollY;
    float           AnimCount;
} EX05_14_STRUCT ;

void init05_14(TCB* thisTCB)
{
    //使用するテクスチャの読み込み
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥MAP_CHIP.png",&g_pTex[0] );
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥MAP_CHIP_B.png",&g_pTex[1] );
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥MAP_CHIP_C.png",&g_pTex[2] );
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥MAP_CHIP_D.png",&g_pTex[3] );
}

void exec05_14(TCB* thisTCB)
{
#define ANIM_SPEED 1.0/8.0      //アニメーションスピード
#define ANIM_COUNT 4            //アニメーション数
```

```
#define CHIP_SIZE 32            //マップチップの大きさ
#define MAP_SIZE_X 20           //マップのサイズX
#define MAP_SIZE_Y 15           //マップのサイズY

unsigned char map_data[ MAP_SIZE_Y ][ MAP_SIZE_X ] =
{//0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19
{  2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, },  //0
{  2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, },  //1
{  2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, },  //2
{  2, 2, 2, 2, 2, 2, 2, 2, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, },  //3
{  2, 2, 2, 2, 2, 2, 2, 2, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, },  //4
{  2, 2, 2, 2, 2, 2, 2, 2, 1, 3, 3, 3, 1, 2, 2, 2, 2, 2, 2, 2, },  //5
{  2, 2, 2, 2, 2, 2, 2, 2, 1, 3, 0, 3, 1, 2, 2, 2, 2, 2, 2, 2, },  //6
{  2, 2, 2, 2, 2, 2, 2, 2, 1, 3, 3, 3, 1, 2, 2, 2, 2, 2, 2, 2, },  //7
{  2, 2, 2, 2, 2, 2, 2, 2, 1, 3, 3, 3, 1, 2, 2, 2, 2, 2, 2, 2, },  //8
{  2, 2, 2, 2, 2, 2, 2, 2, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, },  //9
{  2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, },  //10
{  2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, },  //11
{  2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, },  //12
{  2, 2, 2, 2, 2, 2, 2, 2, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, },  //13
{  2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, },  //14
};

    EX05_14_STRUCT* work = (EX05_14_STRUCT*)thisTCB->Work;
    int loopX;
    int loopY;
    int map_chip_pattern;
    SPRITE2  chip;

    RECT chip_ptn[4] = {
    {  0,   0,  32, 32},
    { 32,   0,  64, 32},
    { 64,   0,  96, 32},
    { 96,   0, 128, 32},
    };

    //アニメーションで切り替えるマップチップのパターンを取得する
    work->AnimCount += ANIM_SPEED;
    //アニメーションカウンタを補正
    if( work->AnimCount >= ANIM_COUNT )work->AnimCount -= ANIM_COUNT;
```

```
    map_chip_pattern = work->AnimCount;

    for( loopY = 0; loopY < SCREEN_HEIGHT / CHIP_SIZE ; loopY++ )
    {
        for( loopX = 0; loopX < SCREEN_WIDTH / CHIP_SIZE ; loopX++ )
        {
            //チップの表示座標を計算
            chip.X = loopX * CHIP_SIZE;
            chip.Y = loopY * CHIP_SIZE;

            //描画するチップのデータを取得
            chip.SrcRect = &chip_ptn[  map_data[ loopY ][ loopX ]  ];

            SpriteDraw( &chip, map_chip_pattern);
        }
    }
}
```

5-15 ラスタースクロール

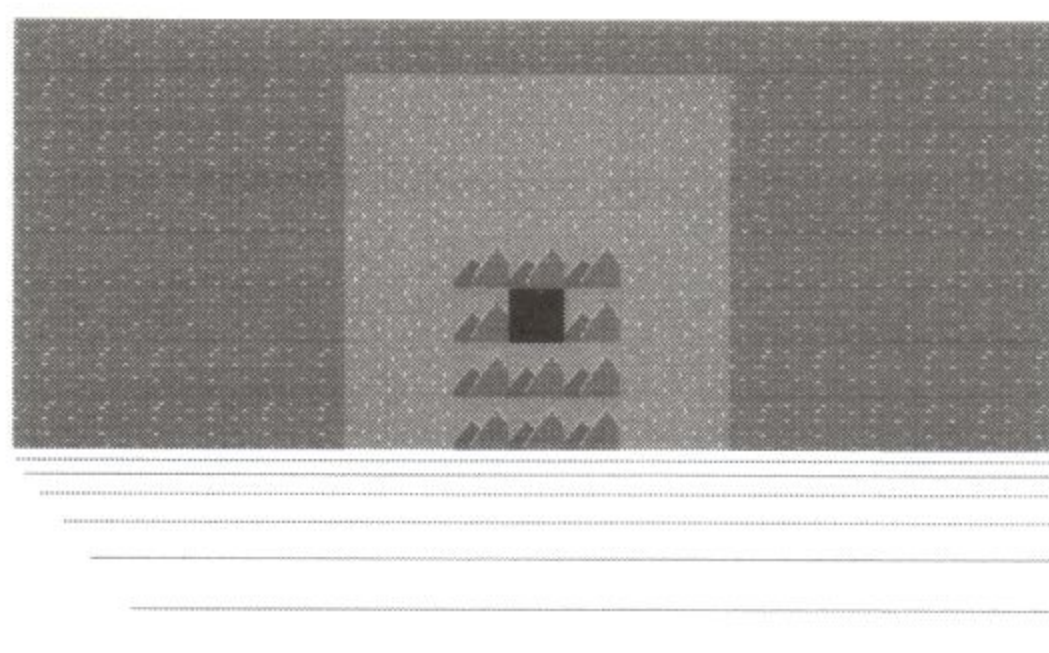
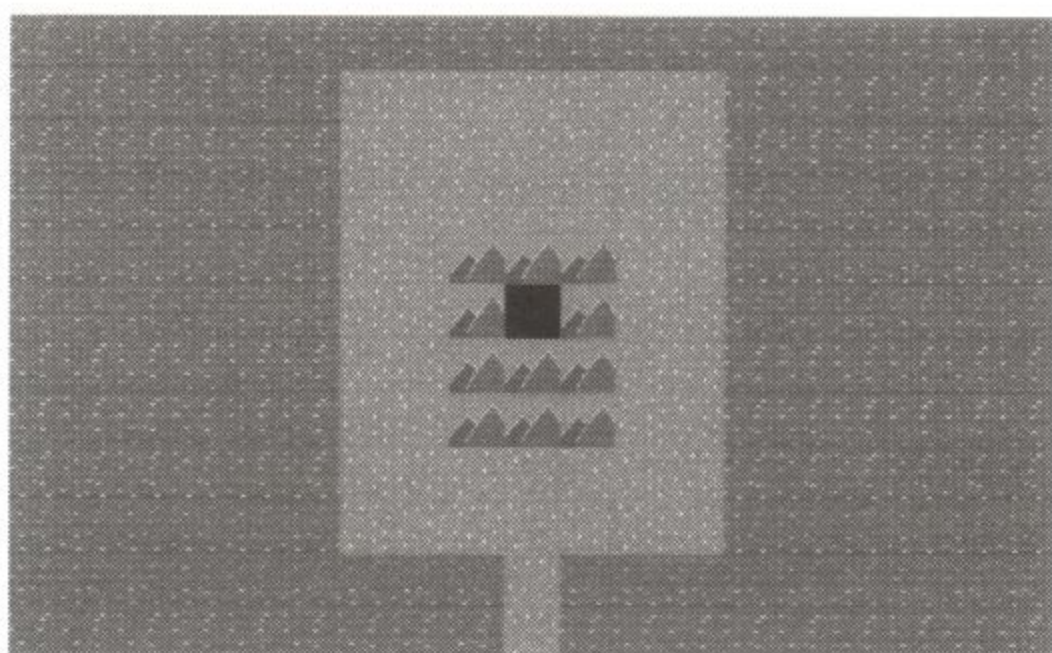
ラスタースクロールとは

ラスタースクロールを実装してみましょう。

ラスタースクロールとは、古くから使われている画像エフェクトで、水平線単位でスクロールを行なう手法です。

DirectX でラスタースクロールを行なうには、画像を水平線単位で分割して描画してやります。

図5 －15－1 ラスタースクロールのイメージ図

絵を水平線単位にばらばらにしてから操作する

ラスタースクロールのプログラミング

実際の表示ですが、それほど難しくはありません。

表示する画像を上から順番に1ラインづつ分割していき、指定されたスクロール座標に表示してやります。

```
void ex5_RasterDraw( short*  raster_pos, int bitmap_id)
{
    int   loop;
    RECT  slice_work = { 0, 0, 1024, 0,};  //分割用のワーク
    D3DXVECTOR3 pos;

    for(loop = 0;loop < SCREEN_HEIGHT; loop++ )
    {    //スクロール座標設定
        pos.x = raster_pos[ loop ];
        pos.y = loop;
```

```
        pos.z = 0;
        //分割
        slice_work.top    = loop;
        slice_work.bottom= loop + 1;
        //描画
        g_pSp->Draw( g_pTex[bitmap_id], &slice_work, NULL, &pos,
0xffffffff);
    }
}
```

実際に扱う際は、スクロール座標を制御することが多いと思いますので、そこの部分も見てみましょう。

LIST 5 －15－1 スクロール座標の制御

```
void exec05_15(TCB* thisTCB)
{
#define RASTER_SIZE  128
    int   loop;
    float raseter_cycle;
    short raster_pos[SCREEN_HEIGHT];

    raseter_cycle = g_Count * 0.0125;
    for( loop = 0; loop < SCREEN_HEIGHT; loop++)
    {
        raseter_cycle += 0.0125;
        raster_pos[ loop ] = sin( raseter_cycle ) * RASTER_SIZE;
    }

    ex5_RasterDraw( raster_pos, 0);
}
```

こちらもさほど難しくはありません。

基本的には、スクロール座標を記録するための配列を用意し、そこにスクロール座標を設定してやるだけです。

今回はエフェクトとして代表的な、サインカーブのデータを周期をずらしながら書き込んでいます。

設定終了後、スクロール座標を格納した配列を、先ほどの描画処理に渡してやれば、描画は終了です。

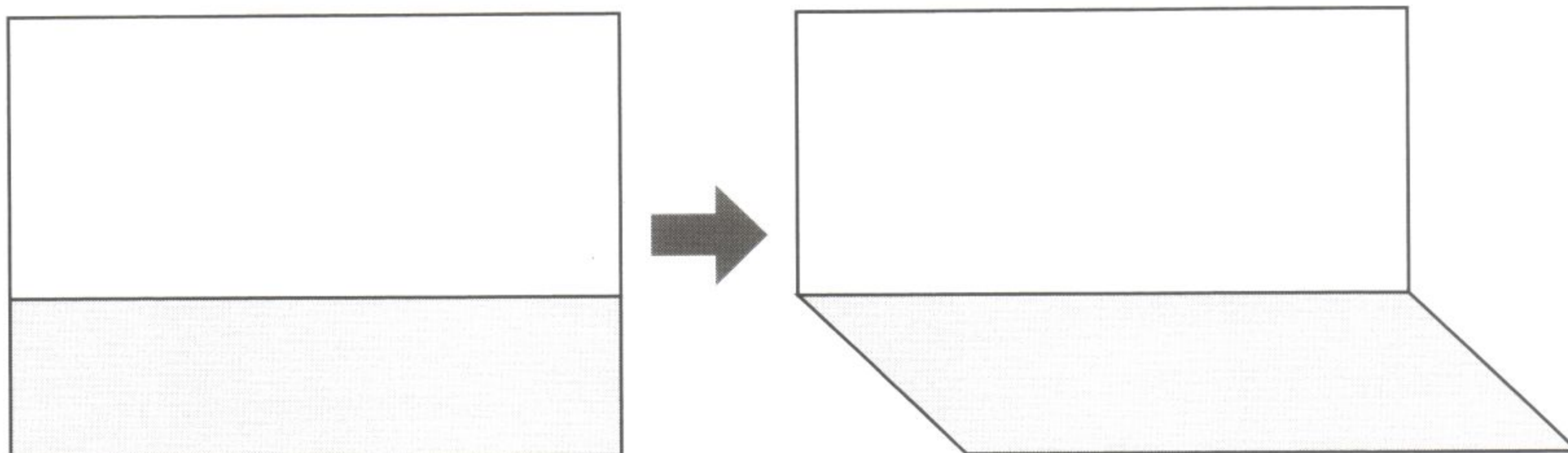
5-16 奥行きのある地面を表示

ラスタースクロールの応用

ラスタースクロールを用いた多重スクロールを行ない、奥行きのある地面を表示してみましょう。

この処理は、対戦格闘ゲームの背景などでよく使われており、立体的に見せるという意味では非常に効果的な画像処理です。

図5−16−1 奥行きのあるラスタースクロールのイメージ図

地面の位置から1 水平線ごとにずらすことによって立体的に見せる

用意する画像が大事

実際の処理ですが、まず、スクロールさせる背景の画像を用意します。

実はこの処理で重要なのは、実はプログラムよりも描画する背景グラフィックの方なのです。

きちんと遠近感のある背景でなければ、立体感のある綺麗なスクロールに見えません。

サンプルでは、立体的な背景という事で分かりやすいワイヤーフレームを用いています。

ラスタースクロールの処理

次にラスタースクロールの処理をする部分ですが、実はやっている事は通常のラスタースクロール処理とほぼ同じです。

地面を分割し、背景の画像に応じて、1水平ライン毎にスクロールさせてやります。

さて、そのスクロールする座標の値ですが、これは難しい3D計算等をする必要はまったくありません。

「スクロールする幅」に、高さをずらした値を掛けてやる事で、簡単に割り出す事ができます。

サンプルプログラム解説

では、実際のプログラムを見ていきましょう。

まず、最初の部分ですが、ここは入力に合わせて背景をスクロールさせる部分です。

ここは特に特殊な事はしておらず、入力キーに合わせて、背景のスクロール値を変更しているだけです。

座標値の計算

次に、実際のラスタースクロールの座標値を計算する部分です。

まず、画面を分割して、背景の表示座標を設定します。これは高さ等に関係なく、すべての座標に設定します。

そして、ラスタースクロールを開始する高さの部分から、スクロールの値を計算して加算していきます。

計算式は、「スクロール幅×計算する高さ÷ラスタースクロールさせる高さ」です。

少し分かりにくいかもしれませんが、スクロールする幅を、高さに応じてどの程度ずらすかを計算している、と考えるとイメージしやすいでしょうか。

最後に計算した座標を元に、背景を表示してやれば終了です。

LIST 5 ―16―1 奥行きのある地面を表示

```
void init05_16(TCB* thisTCB)
{
	//使用するテクスチャの読み込み
	D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥PERS_BG.png",&g_pTex[0] );
}

void exec05_16(TCB* thisTCB)
{
#define RASTER_START	256		//ラスタースクロールを開始する位置
#define RASTER_END	512		//ラスタースクロールを終了する位置
#define BACK_POS	180
#define SCROLL_SPEED	4
typedef struct{
	int		ScrollPos;
	short		RasterPos[SCREEN_HEIGHT];
```

```
} EX5_16_STRUCT ;

    EX5_16_STRUCT* work = (EX5_16_STRUCT*)thisTCB->Work;
    int   loop;

    //左右へスクロール
    if(!(g_InputBuff & KEY_LEFT))
    {
        work->ScrollPos += SCROLL_SPEED;
        if( work->ScrollPos >  BACK_POS ) work->ScrollPos = BACK_POS;
    }

    if(!(g_InputBuff & KEY_RIGHT))
    {
        work->ScrollPos -= SCROLL_SPEED;
        if( work->ScrollPos < -BACK_POS ) work->ScrollPos = -BACK_POS;
    }

    //奥行き部分を分割
    for( loop = 0; loop < SCREEN_HEIGHT; loop++)
    {   //地面以外の部分は固定位置で、スクロールしない
        work->RasterPos[ loop ] = -BACK_POS;

        //地面部分をスクロール座標にあわせて歪ませる
        if( loop >= RASTER_START )
        {
            work->RasterPos[ loop ] += work->ScrollPos * (loop /
(float)(RASTER_END - RASTER_START) );
        }
    }

    ex5_RasterDraw( work->RasterPos, 0);
}
```

5-17 フェードイン・アウトを行なう

半透明を使用したフェードイン・アウト

フェードイン・アウトの処理を作成してみます。

この処理はステージの切り替え時や、ロード中など、ゲームの種類を問わず、かなりの場所で使われています。

ところが実はこの処理、その実現方法が、かなり多岐にわたっており、全部を紹介するのは困難です。

そのため今回は、その中でも汎用性があり、また結構手軽な方法という事で、半透明を使用したフェードイン・アウトを紹介します。

フェードアウトの表現方法

さて、その手法ですが、まずフェードアウト（元画像がだんだん隠れていく処理）を考えてみましょう。

まず、フェードの元となる画像があります。そしてそこに、完全に透明にした、黒い画像を重ねてみます。

当然、黒い画像とはいえ透明ですので、最初は見えません。

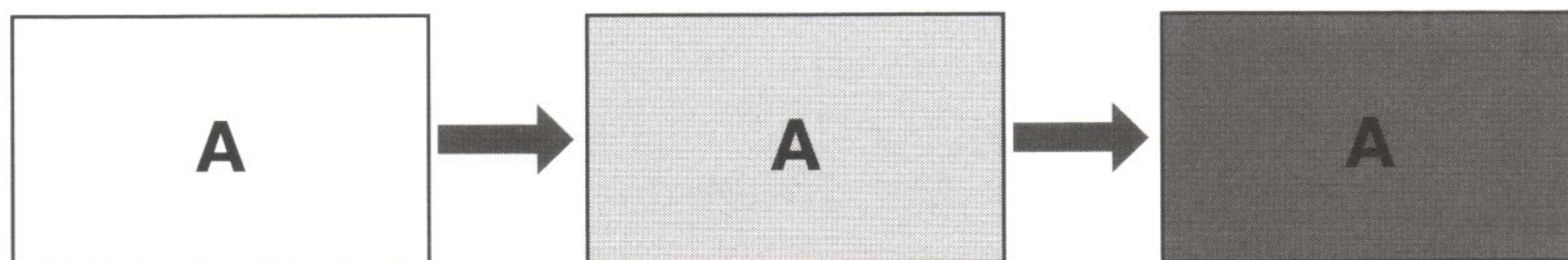
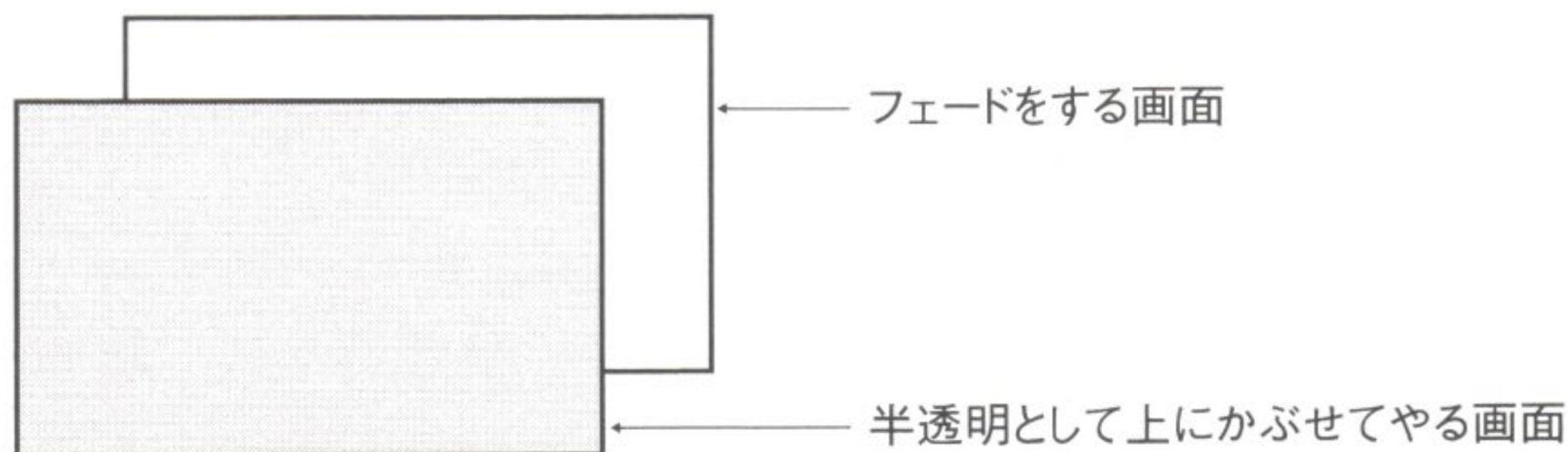
これを徐々に、「不透明」にしてやることで、「黒い画面」に近づけてやります。

図5－17－1 半透明を使用したフェード処理のイメージ図

画面に半透明の画像をかけてやり、フェードを表現する

フェードをする画面

半透明として上にかぶせてやる画面

A → A → A

かぶせる画面の透明度を徐々に変えてやる

イメージが伝わりましたでしょうか？

これを、「黒い画面」から透明にしてやれば、フェードインの処理になります。

また、この処理は、データを変更すれば、色々と応用が利きます。

例えば、白い画面を使えば、ホワイトイン・アウトになりますし、別の背景画像を用いれば、複数の画面を切り替えるクロスフェードになります。

フェードイン・アウトのプログラミング

では、実際のプログラムを見ていきます。

サンプルは背景の画像をＺキーでフェードイン、Ｘキーでフェードアウトさせるものです。

初期化

まず初期化ですが、２枚の画像、フェード元の画像とフェード用の画像を読み込みます。

次にメイン処理ですが、まずフェード中かどうかで処理を分けます。

フェード中でなければ、キー入力をチェックし、フェードインかアウト、どちらかの状態に移行します。

フェードの状態は、タスク上での変数FadeModeで管理しており、フェードイン／フェードアウト／フェード無しの３つの状態を保持します。

フェード処理

さてフェード処理ですが、これは、変数FadeCountの数値を管理する事で行ないます。

この数値が、実質的に半透明の「透明度」を表しています。変数名がCountなのは、この変数で時間もフェードの時間もカウントしているためです。

処理はイン・アウト両方の場合でケースが分かれており、インの場合は、だんだん透明になるように、アウトの場合は、だんだん不透明になるようになっています。

そして最終的に、完全に透明、又は不透明になった時点でフェード処理は終了します。

フェードの描画処理の応用

最後に描画処理について触れておきます。

フェード用の画像は専用の関数exec05_17_FadeDrawで描画されています。

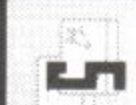
これは引数に、「半透明のレベル」を渡せるようにした関数で、内部的には通常の背景描画関数とほぼ同じです。

したがって、ここを変更することで、背景以外でもいろんな物をフェードの対象にする事ができます。

LIST 5 －17－1

```
typedef struct{
    BACK_GROUND     BaseBg;
    BACK_GROUND     FadeBg;
    int             FadeMode;           //0.フェード無し、1.フェードアウト -1.フェードイン
    int             FadeCount;
} EX05_17_STRUCT ;

void exec05_17_FadeDraw(BACK_GROUND* bg,int bitmap_id,unsigned int
fade_level)
{
    D3DXVECTOR3 pos;
    DWORD       color;

    pos.x = bg->X;
    pos.y = bg->Y;
    pos.z = 0;
    //フェードレベルによってα値と色を設定
    color = D3DCOLOR_ARGB( fade_level ,0xff,0xff,0xff );

    g_pSp->Draw( g_pTex[bitmap_id],NULL,NULL,&pos, color);
}


void init05_17(TCB* thisTCB)
{
    //使用するテクスチャの読み込み
    D3DXCreateTextureFromFile( g_pD3Ddevice, "..¥¥..¥¥data¥¥bg01.png",
&g_pTex[0] );
    D3DXCreateTextureFromFile( g_pD3Ddevice, "..¥¥..¥¥data¥¥change_bga.png",
&g_pTex[1] );
}

void exec05_17(TCB* thisTCB)
{
#define FADE_IN   -1
#define FADE_OUT  1
#define FADE_NONE 0
#define FADE_SPEED 4

```

```
    EX05_17_STRUCT* work = (EX05_17_STRUCT*)thisTCB->Work;

    //フェード中かどうかで処理を分ける
    if( work->FadeMode == FADE_NONE )
    {//キー入力でフェードの開始
        if( g_DownInputBuff & KEY_Z )
        {//フェードイン処理開始
            work->FadeMode  = FADE_IN;
            work->FadeCount = 0xff;
        }
        if( g_DownInputBuff & KEY_X )
        {//フェードアウト処理開始
            work->FadeMode  = FADE_OUT;
            work->FadeCount = 0;
        }
    }else{
    //フェードイン・アウト中
        if( work->FadeMode == FADE_IN )
        {//フェードイン処理
            work->FadeCount -= FADE_SPEED;
            if( work->FadeCount <= 0 )
            {
                work->FadeMode  = FADE_NONE;
                work->FadeCount = 0;
            }
        }
        if( work->FadeMode == FADE_OUT )
        {//フェードアウト処理
            work->FadeCount += FADE_SPEED;
            if( work->FadeCount >= 0xff )
            {
                work->FadeMode  = FADE_NONE;
                work->FadeCount = 0xff;
            }
        }
   }

    //対象の背景を表示
    BGDraw( &work->BaseBg, 0);
```

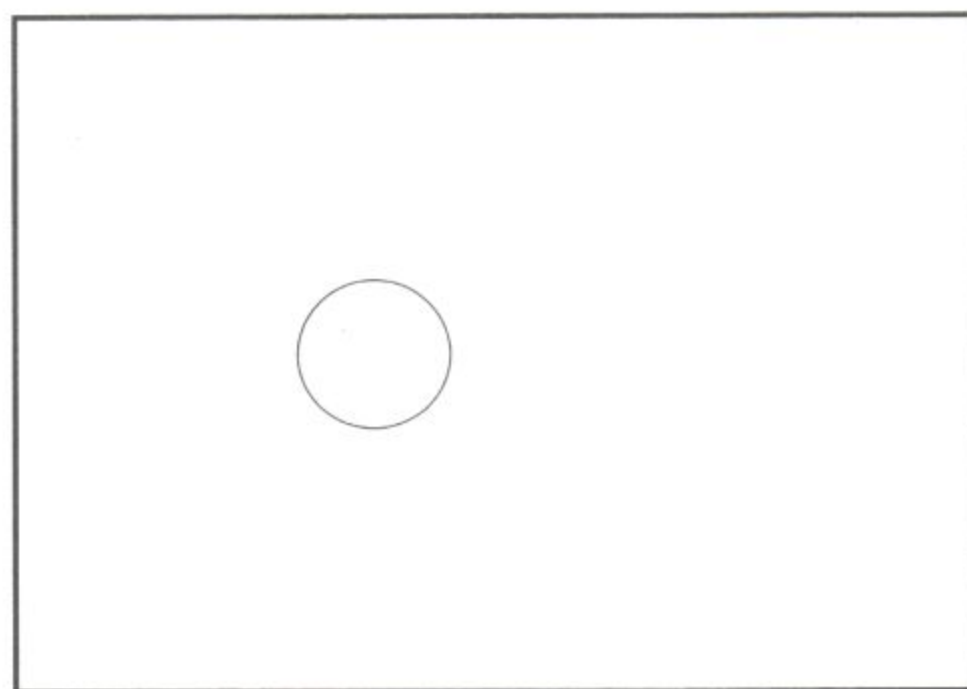
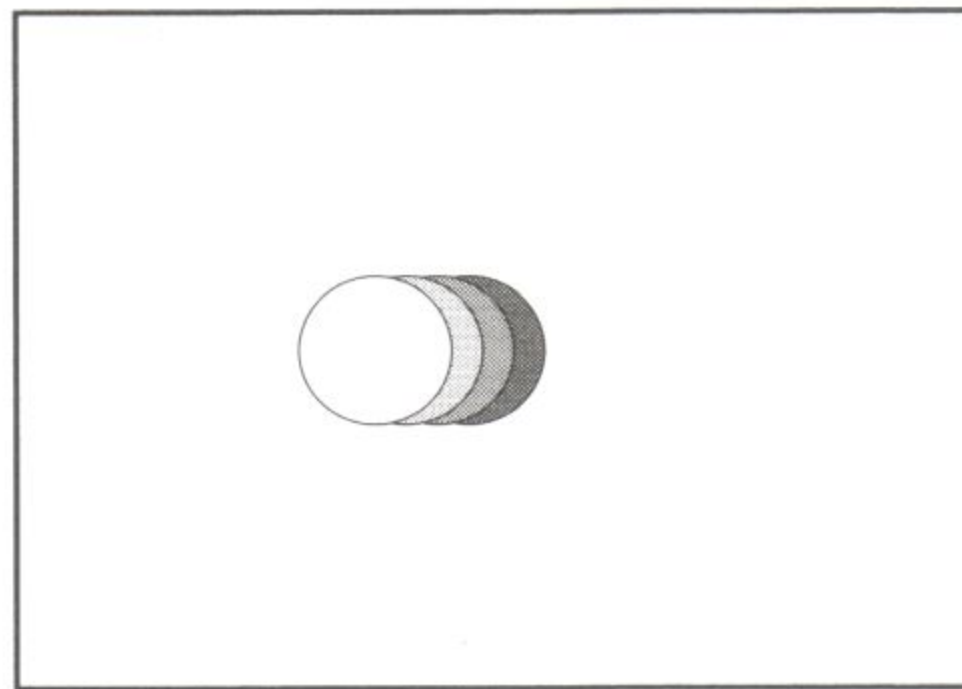
```
	//フェード中はフェード用の画像を表示
	if( work->FadeMode != FADE_NONE ) exec05_17_FadeDraw( &work->FadeBg,
1,work->FadeCount );
}
```

5-18 キャラクターに残像を付ける

残像の使いどころ

キャラクターに残像を付ける処理を作成してみます。

残像処理は、主に演出として使われ、パワーアップの瞬間やボスの移動等、特定の瞬間やキャラクターを目立たせたい時に使用される事が多いようです。

また、残像を残すキャラクターを画面上から見失いにくくする効果もあるため、高速移動を行なう自機等に使用される事もあります。

図5 ―18―1 残像を行なうとキャラが増えるため目立ちやすくなるイメージ図

残像無しの場合

残像有りの場合

処理方法

さてその処理方法ですが、キャラクターの座標を、残しておきたい残像数だけ記録バッファに記録しておき、描画時にはその記録座標にもキャラクターを描画してやります。

この際、そのまま描画すると元のキャラクター区別がつかないため、残像の数に合わせて色を薄くするようにします。

残像のプログラミング

それでは、サンプルを見て行きましょう。サンプルは、方向キーで自機を動かし、Zキーで残像の数を変更するものです。

座標の移動

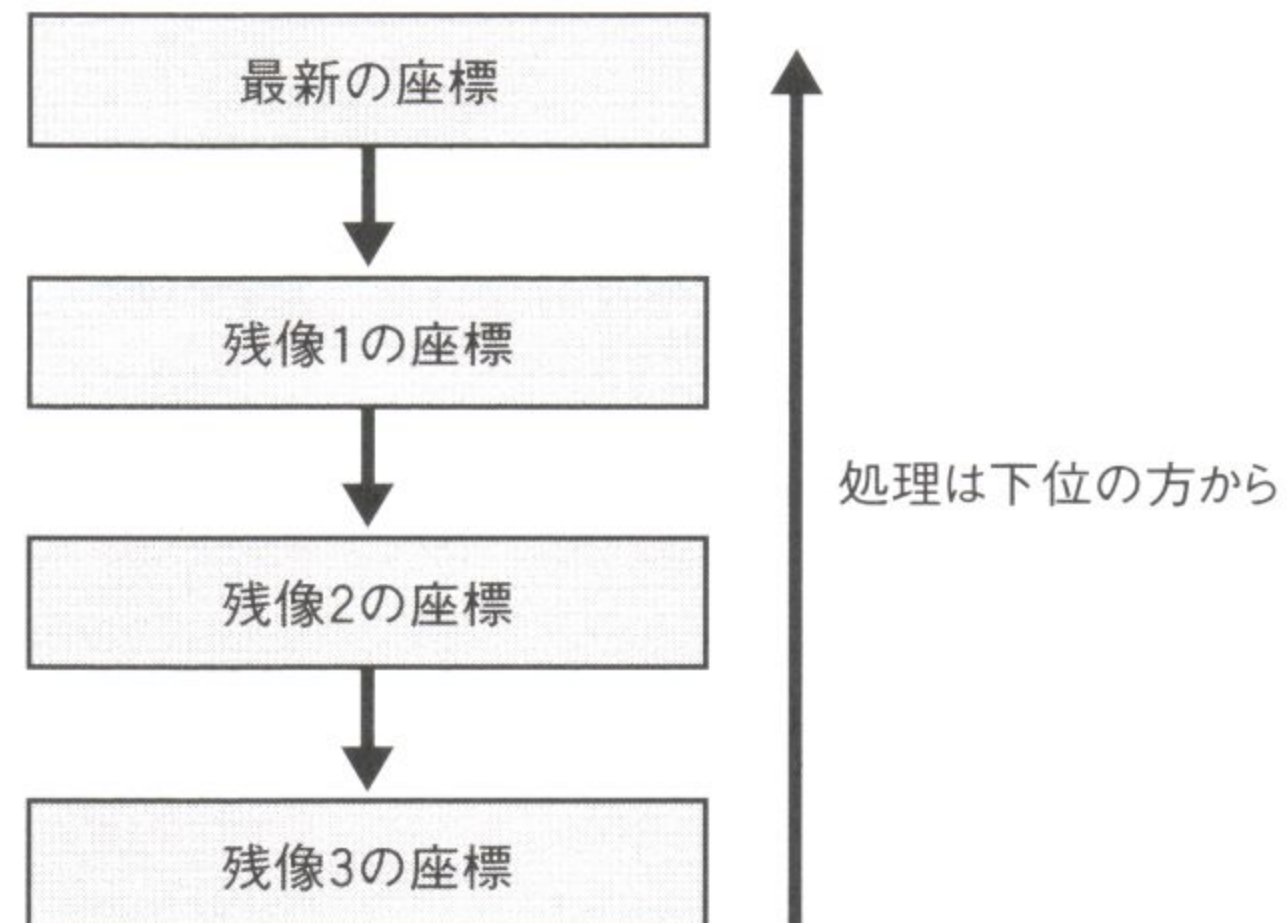
まずはじめに、残像座標の位置移動を行ないます。これは、もし先に記録を行なってしまうと、バッファの上書きが行なわれてしまうためです。

座標の移動は古い順に座標をコピーする事で行なわれ、終了後に、自機の最新の座標をバッファの先頭に書き込みます。

図5 −18−2 バッファを古い順にコピーするイメージ図

1つ前の座標バッファを次々とコピーしていく
処理の順番は古い方(下の方)から行なう

最新の座標
↓
残像1の座標
↓
残像2の座標
↓
残像3の座標

処理は下位の方から

残像の数、表示処理

次に、残像の数を変更する処理を行なっています。ここは、単純な加算処理ですので問題は無いでしょう。

その後、残像の表示処理を行なっています。まず、残像の色の濃さを決定するため、残像数に合わせた色の加算値を計算しています。計算は濃度の最大数(0xff)を残像の数で割ってやるだけです。

そして、残像数分の表示処理を行ないます。色の濃さは先ほど計算した値を累積加算して行き、専用のスプライト描画関数EX05_18_CustumDrawに渡してやります。

この関数は、引数として色の濃さを受け取り、その濃度で描画を行なう物です。

残像の表示の順番

なお、描画する時は、必ず残像の末尾から表示してやります。こうしないと、最初の残像が最後の残像に上書きされてしまうため、非常に違和感のあるものになってしまうからです。

最後に、自機を描画して残像処理は完了です。

LIST 5 －18－1 キャラクターに残像を付ける

```
#define IMAGE_COUNT 8
#define MOVE_SPEED  8.0

typedef struct{
    SPRITE          Sprt;
    SPRITE          Image;
    D3DXVECTOR2     PosBuff[ IMAGE_COUNT ];  //残像の座標
    int             ImageCount;
} EX05_18_STRUCT;

void EX05_18_CustomDraw(SPRITE* pspr,int bitmap_id,int color_depth)
{
    D3DXVECTOR3 pos;

    pos.x = pspr->X;
    pos.y = pspr->Y;
    pos.z = 0;

    color_depth =
        D3DCOLOR_ARGB( 0xff,color_depth,color_depth,color_depth );
    g_pSp->Draw( g_pTex[bitmap_id],NULL,NULL,&pos, color_depth);
}

void init05_18(TCB* thisTCB)
{
    //使用するテクスチャの読み込み
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥0044.png",&g_pTex[0] );
}

void exec05_18(TCB* thisTCB)
{
    EX05_18_STRUCT* work = (EX05_18_STRUCT*)thisTCB->Work;
    int loop;
    float add_color_depth;
    float color_depth = 0;

    //座標履歴(残像の表示座標)をずらす
    for( loop = IMAGE_COUNT-1; loop > 0; loop-- )
```

```
        {//一番古い座標から消していく
            work->PosBuff[ loop ].x = work->PosBuff[ loop-1 ].x;
            work->PosBuff[ loop ].y = work->PosBuff[ loop-1 ].y;
        }

        //キー入力による移動
        if( g_InputBuff & KEY_UP    ) work->Sprt.Y -= MOVE_SPEED;
        if( g_InputBuff & KEY_DOWN  ) work->Sprt.Y += MOVE_SPEED;
        if( g_InputBuff & KEY_RIGHT ) work->Sprt.X += MOVE_SPEED;
        if( g_InputBuff & KEY_LEFT  ) work->Sprt.X -= MOVE_SPEED;

        //自機の移動座標を履歴に記録する
        work->PosBuff[ 0 ].x = work->Sprt.X;
        work->PosBuff[ 0 ].y = work->Sprt.Y;

        //残像の表示数を変更
        if( g_DownInputBuff & KEY_Z )
        {//最大表示数を越えないように補正
           if( ++work->ImageCount >= IMAGE_COUNT ) work->ImageCount = 0;
        }

        //残像の色の濃さを計算する加算値を取得
        if(work->ImageCount != 0)
        {
           add_color_depth = 0xff / work->ImageCount;
        }

        //指定した数だけ残像を表示する
        for( loop = work->ImageCount; loop > 0; loop-- )
        {
            //座標履歴から表示座標を取得
            work->Image.X = work->PosBuff[ loop ].x;
            work->Image.Y = work->PosBuff[ loop ].y;
            //残像の色の濃さを計算
            color_depth += add_color_depth;
    //色の濃さを反映する専用の描画関数で描画
            EX05_18_CustomDraw( &work->Image,0,(int)color_depth );
        }
    SpriteDraw( &work->Sprt, 0);
    }
```

5-19 キャラクターの影を付ける

キャラクターの影の意味

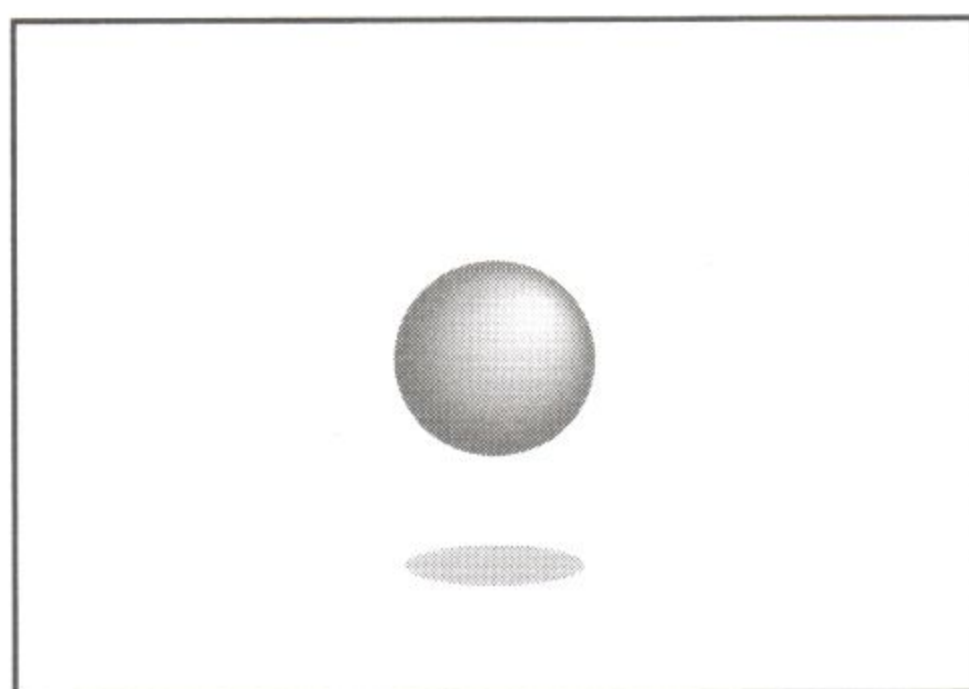
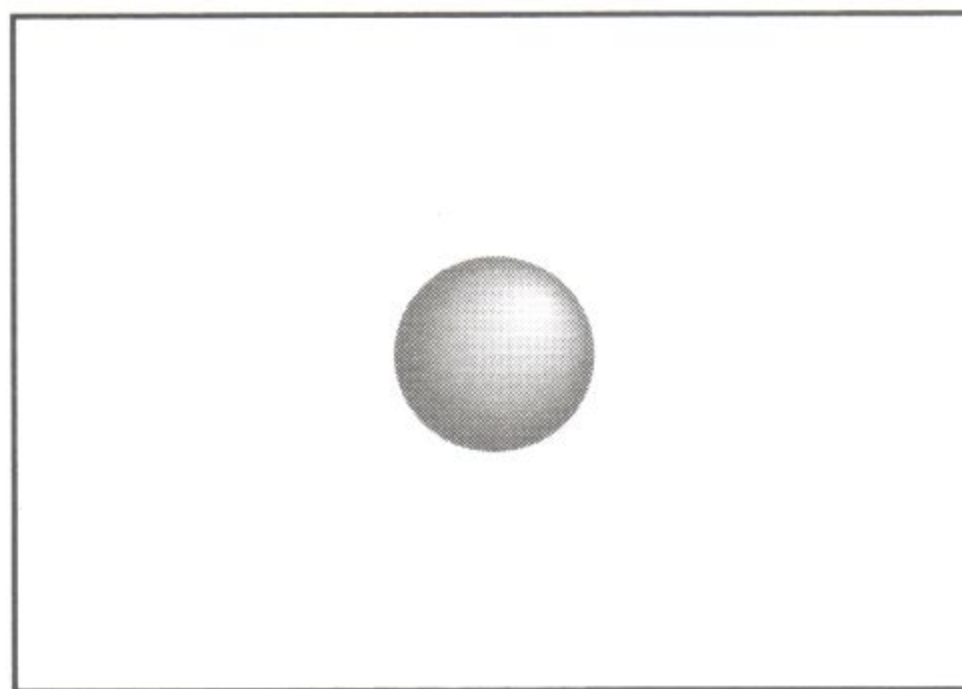
アクションゲームなどで、キャラクターに影を付ける処理を行なってみます。

実はこの処理自体はさほど難しい事はありません。けれどもゲーム上では２つの重要な意味を持ちます。

１つは、デザイン上の違和感です。実際に背景などと同時に表示をしてみればわかるのですが、影の無いキャラクターは存在感が薄く、どうしてものっぺりとした見栄えのしないものになってしまいます。

もう一つは、プレイヤーにとっての位置関係の把握です。例えばアクションゲーム等でのジャンプ中ですが、実はプレイヤーは無意識に地面の位置を、影でも判断しているのです。

そのため影が無いと、着地の位置などが分かりにくくなり、結果として非常に操作性の悪いゲームの印象を与えてしまうのです。

図５ －19－１ 影の有無を比較する２枚のイメージ図

影がある場合

影が無い場合

実際の影表示のプログラム

では実際に影の処理プログラムを見ていきましょう。

サンプルは、方向キーでキャラクターを動かし、Ｚキーでジャンプを行なうものです。

影の処理自体は単純で、キャラクターの移動にあわせて、Ｘ座標を追従するだけです。

この時Ｙ座標は、想定する地面の位置にあわせます。

もし地面の高さが変わる場合は、この座標を地面に高さに合わせて変えてやる必要があります。

あとは、その座標に影の表示を行うだけです。

なお、処理のはじめにある移動とジャンプの処理ですが、この処理は[6－17]で解説しているためここでは割愛します。

LIST 5－19－1 キャラクターに影を付ける

```
typedef struct{
    SPRITE          Human;
    SPRITE          Shadow;
    float           Acc;
    int             JumpFlag;
} EX05_19_STRUCT;
#define MOVE_SPEED 4.0              //移動速度
#define JUMP_POWER 10.0             //ジャンプ力
#define GRAVITY    0.35             //重力値
#define GRAND_LINE SCREEN_HEIGHT/2  //地面位置
#define CHR_HEIGHT 104.0            //キャラの高さ

void init05_19(TCB* thisTCB)
{
    EX05_19_STRUCT* work = (EX05_19_STRUCT*)thisTCB->Work;

    //使用するテクスチャの読み込み
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥human.png",&g_pTex[0] );
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥shadow.png",&g_pTex[1] );

    //座標の初期化
    work->Human.X = SCREEN_WIDTH / 2;
    work->Human.Y = GRAND_LINE - CHR_HEIGHT;
}

void exec05_19(TCB* thisTCB)
{
    EX05_19_STRUCT* work = (EX05_19_STRUCT*)thisTCB->Work;

    //キャラクターの移動処理
    if( g_InputBuff & KEY_RIGHT ) work->Human.X += MOVE_SPEED;
    if( g_InputBuff & KEY_LEFT  ) work->Human.X -= MOVE_SPEED;
    //ジャンプ処理
```

```
    if( !work->JumpFlag )
    {//着地時の処理
        if( g_DownInputBuff & KEY_Z     )
        {//キー入力でジャンプ
            work->Acc = -JUMP_POWER;
            work->JumpFlag = true;
        }
     }else{
      //ジャンプ中処理
        work->Acc += GRAVITY;
        work->Human.Y += work->Acc;
        if( work->Human.Y > GRAND_LINE - CHR_HEIGHT)
        {//ジャンプ処理終了
            work->Human.Y =  GRAND_LINE - CHR_HEIGHT;
            work->JumpFlag = false;
        }
    }

    //影の処理
    //キャラクターのX座標のみ追従する
    work->Shadow.X = work->Human.X;
    //Y座標は地面に合わせる(ここでは固定値)
    work->Shadow.Y = GRAND_LINE;

    //影の描画を先に行なう
    SpriteDraw( &work->Shadow, 1);
    SpriteDraw( &work->Human, 0);
}
```

5-20 マウスなどでスクロールさせる１

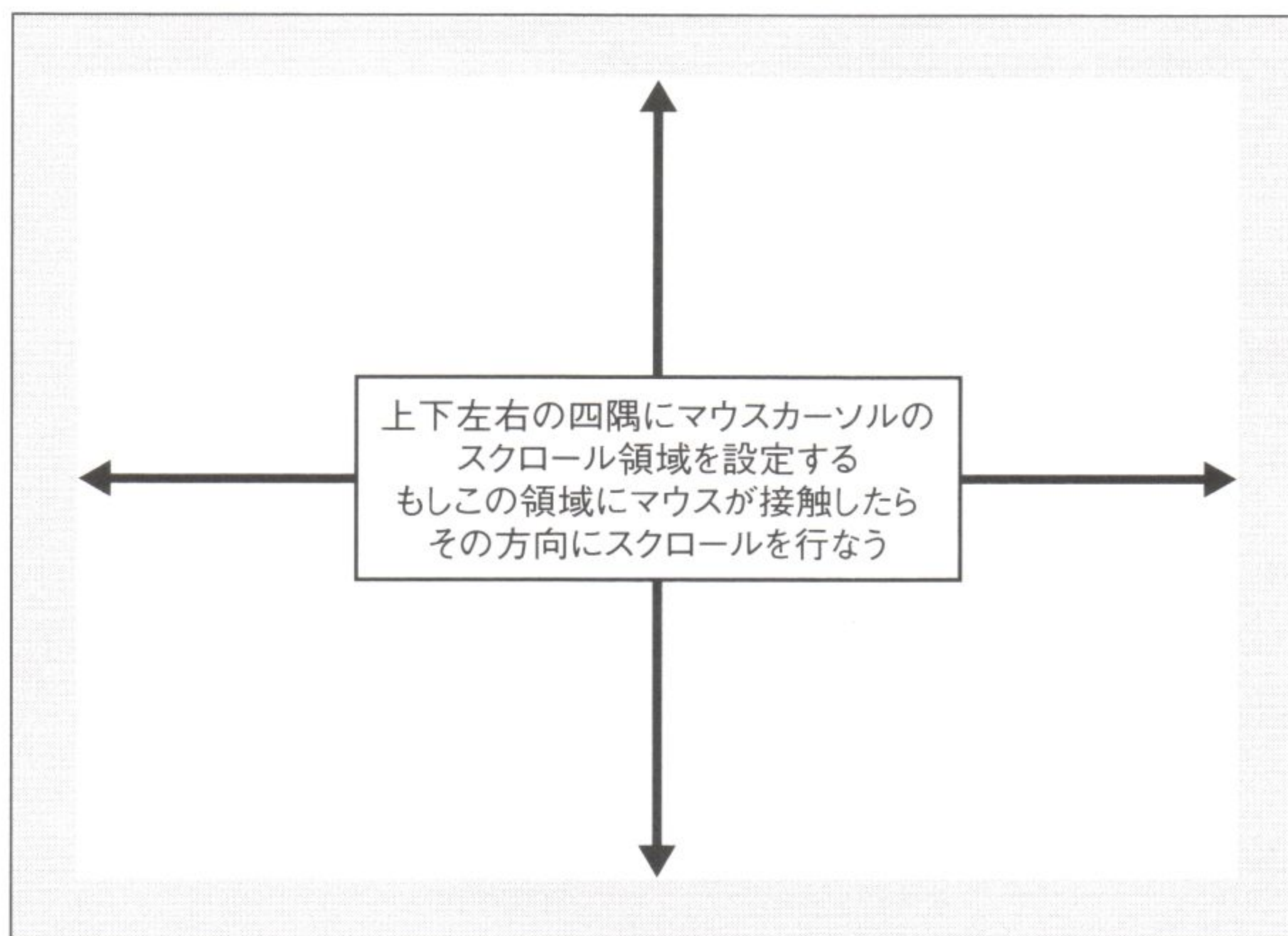
画面端にカーソルを合わせてスクロール

ここでは、マウスを使って画面を自由にスクロールさせる方法を紹介します。

マウスでのスクロール方法はゲームによって様々ですが、よくある手法としては、画面の端にマウスカーソルを移動させるとスクロールする、というものです。

処理の方法ですが、画面の上下左右４隅にマウスカーソルに反応する領域を設けます。

この４箇所の領域にマウスが乗っかっていれば、対応した方向に対してスクロールをします。

図５ －20－１ 画面の四隅に領域を設けるイメージ図

スクロールのプログラミング

実際のプログラムを見ていきましょう。まず、ここで重要なのはデータです。

マウスが反応する領域を指定するデータと、それに対するスクロール値を１つの構造体にまとめ、データを定義しています。

```
typedef struct{
    int             AddX;       //マウスに反応した時加算するXの値
    int             AddY;       //マウスに反応した時加算するYの値
    RECT            MoveRect;   //マウスが反応する領域
} EX05_20_DATA;

    EX05_20_DATA  MovePoint[] =
{//     X     Y     RECT
    {  0,    4, {    0,    0, 640,   16,},},         //1つ目 上
    {  0,   -4, {    0, 464, 640, 480,},},           //2つ目 下
    { -4,    0, { 624,    0, 640, 480,},},           //3つ目 右
    {  4,    0, {    0,    0,   16, 480,},},         //4つ目 左
};
```

データは配列で用意されており、このデータを調整すれば画面端だけではなく、様々な色んな位置にあわせたマウス移動に対応できます。

データが決まれば、後の処理はさほど難しくありません。

まず、マウスの座標と、データ中の領域を定義している部分とを比較します。

もしマウスが領域の中に入っていれば、比較した領域と同じデータ中にある、スクロール座標を加算します。

これを、定義されたデータの個数分繰り返してやればOKです。

あとは、スクロール座標が表示の範囲外になっていないかをチェックし、表示を行なって終了です。

LIST 5 －20－1 マウスなどで自由にスクロールさせる

```
typedef struct{
    BACK_GROUND     BG;
} EX05_20_STRUCT;

typedef struct{
    //マウスに反応した時加算するXの値
    int             AddX;
    //マウスに反応した時加算するYの値
    int             AddY;
    //マウスが反応する領域
```

```
    RECT              MoveRect;
} EX05_20_DATA;

void init05_20(TCB* thisTCB)
{
    //使用するテクスチャの読み込み
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥0022.png",&g_pTex[0] );
}

void exec05_20(TCB* thisTCB)
{
//マウスを移動させるポイントの数
#define POINT_COUNT 4

    EX05_20_STRUCT* work = (EX05_20_STRUCT*)thisTCB->Work;
    EX05_20_DATA  MovePoint[] =
{//     X     Y     RECT
    {  0,   4, {   0,   0, 640,  16,},},        //1つ目 上
    {  0,  -4, {   0, 464, 640, 480,},},        //2つ目 下
    { -4,   0, { 624,   0, 640, 480,},},        //3つ目 右
    {  4,   0, {   0,   0,  16, 480,},},        //4つ目 左
};
    int loop;

    //マウスカーソルに反応するポイントの数だけループ
    for( loop = 0; loop < POINT_COUNT ;loop++ )
    {//マウスカーソルが反応する領域をチェック
        if( g_MousePos.y > MovePoint[ loop ].MoveRect.top    &&
            g_MousePos.y < MovePoint[ loop ].MoveRect.bottom &&
            g_MousePos.x > MovePoint[ loop ].MoveRect.left   &&
            g_MousePos.x < MovePoint[ loop ].MoveRect.right  )
        {//領域内にあれば、座標を加算
            work->BG.X += MovePoint[ loop ].AddX;
            work->BG.Y += MovePoint[ loop ].AddY;
        }
    }

    //スクロールを画面の範囲内に収める
    if( work->BG.X > 0    ) work->BG.X = 0;
```

```
    if( work->BG.X < -640 ) work->BG.X = -640;
    if( work->BG.Y > 0    ) work->BG.Y = 0;
    if( work->BG.Y < -32  ) work->BG.Y = -32;

    BGDraw(&work->BG,0);
}
```

5-21 マウスなどでスクロールさせる２

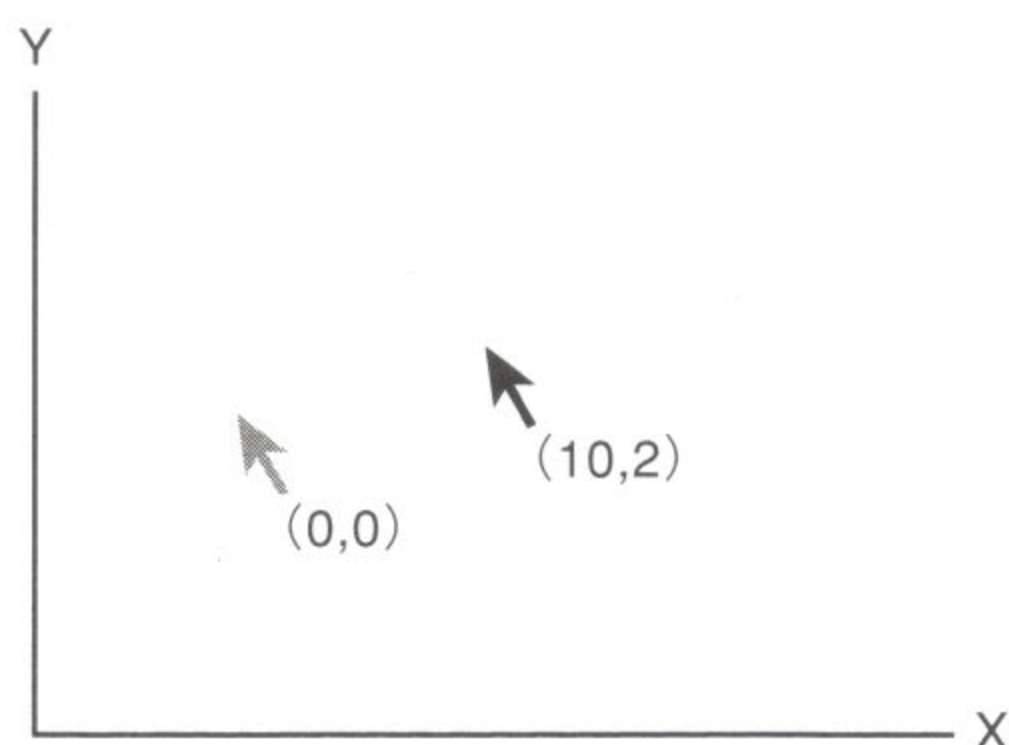
マウスドラッグによる画面スクロール

前項に引き続き、マウスのドラッグによる画面のスクロール方法を紹介します。

処理の方法ですが、マウスボタンが押されている間、マウスが移動した分量、すなわち移動量を算出します。

その後、算出した値をスクロールの座標に加算してやればOKです。

その移動量ですが、これは現在のマウス座標から、１フレーム前のマウス座標を引いてやれば得ることが出来ます。

図5 －21－1　１フレーム前のマウス座標から移動量を算出するイメージ図

1フレーム前の座標を原点として、座標を算出すれば
移動量が求まる

ドラッグによるスクロールのプログラミング

では、実際のプログラムを見ていきましょう。

はじめにドラッグをしているかどうかを確認するため、マウスボタンをチェックします。

ボタンが押されている間は、ドラッグしていると判断できますので、移動量を算出し、スクロールの値として背景の座標に加算します。

その後、次のフレームで使用するため、現在のマウス座標を保存しておきます。

最後は前項と同様に、スクロールの座標が表示の範囲外に出ないように処理をしています。

LIST 5 －21－1 マウスなどで自由にスクロールさせる2

```
typedef struct{
    BACK_GROUND     BG;
    POINT           MouseOldPos;
} EX05_21_STRUCT;

void init05_21(TCB* thisTCB)
{
    //使用するテクスチャの読み込み
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥0022.png",&g_pTex[0] );
}

void exec05_21(TCB* thisTCB)
{
    EX05_21_STRUCT* work = (EX05_21_STRUCT*)thisTCB->Work;


    //マウス左ボタンをチェック
    if( g_MouseButton & MOUSE_L )
    {//ドラッグ中の処理
        //マウスの移動量を算出、スクロール値として加算する
        work->BG.X += g_MousePos.x - work->MouseOldPos.x;
        work->BG.Y += g_MousePos.y - work->MouseOldPos.y;
    }
    //1フレーム前のマウス位置を保存
    work->MouseOldPos = g_MousePos;

    //スクロールを画面の範囲内に収める
    if( work->BG.X > 0    ) work->BG.X = 0;
    if( work->BG.X < -640 ) work->BG.X = -640;
    if( work->BG.Y > 0    ) work->BG.Y = 0;
    if( work->BG.Y < -32  ) work->BG.Y = -32;

    BGDraw(&work->BG,0);
}
```

Chapter 6

移動

逆引き ゲームプログラミング

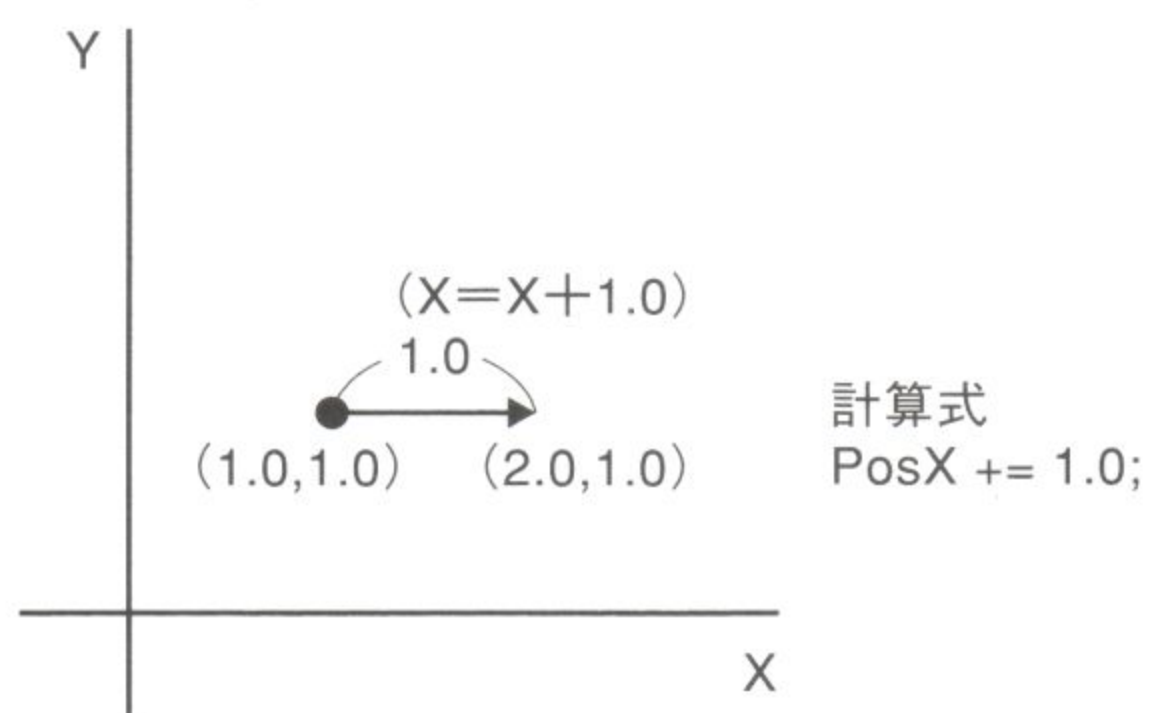
Game programming

6-1　直線的に移動させる

水平、垂直に移動する

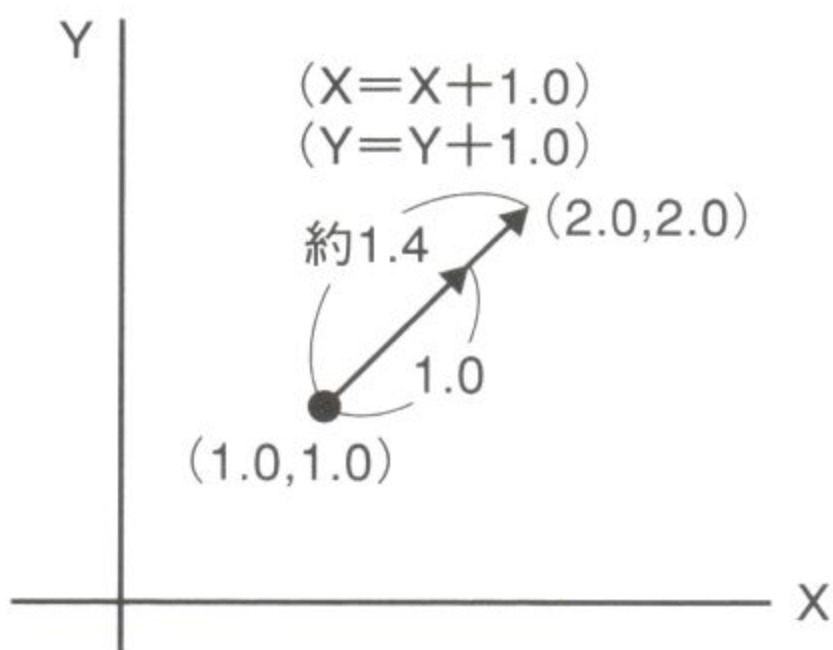
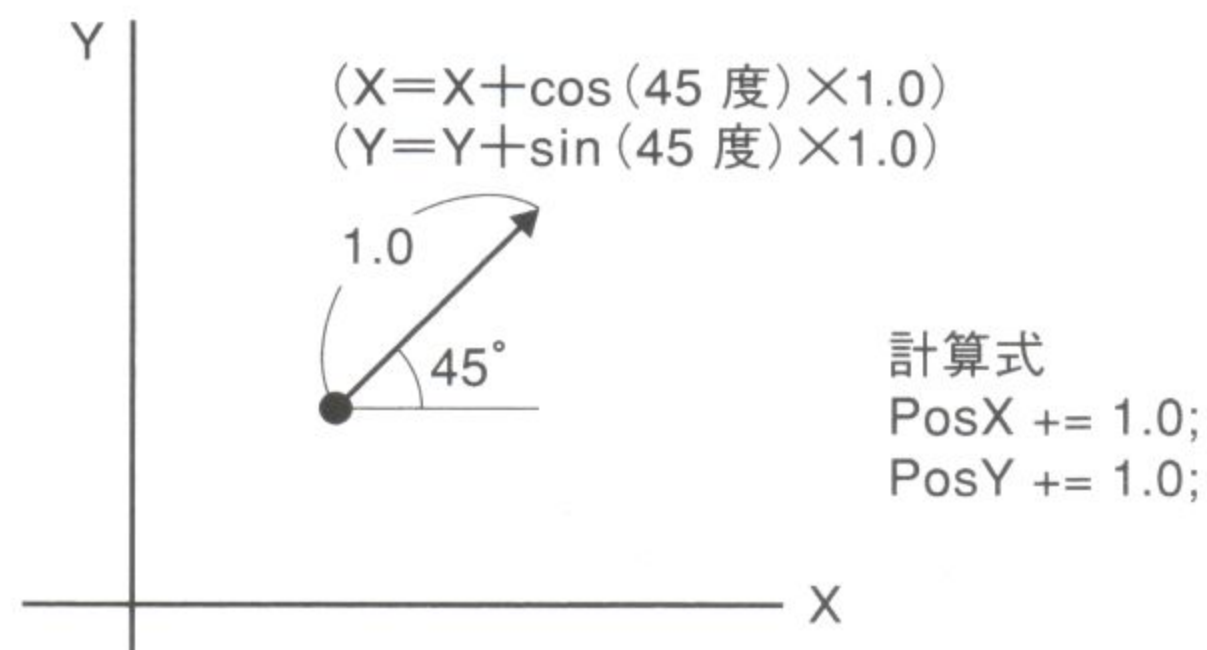
直線的な移動といっても、X軸やY軸にそって移動させるだけであれば、そのまま座標を加算するだけです。

例えば、1フレームに1ドットだけX軸方向に物体を移動させる場合などは、動かす場合は、以下の様な感じになるでしょう。

図6－1－1 軸に対して直線的に移動させるイメージ図と計算式

軸方向のみに1 .0を足せば1.0だけ移動する

斜めに移動するときの注意点

ところが好きな方向に移動、例えば45度方向に移動させるときには、移動速度をそのまま加算してしまうと斜めだけ速度が変化してしまいます。

図6－1－2　45度方向に＋1をすると、速度が変化してしまうイメージ図と計算式

両軸に1.0を足すと、1.0だけ動くのではなく
それ以上動いてしまうので、結果として速度が速くなる

そこで、方向と移動量から加算する数値を求める

これを解決するには、方向と移動量の概念を用います。方向とは文字通り、移動する方向のことで、キャラクターが何処に移動するかを角度で表したものです。

また、移動量とは、移動したい方向に応じて、X座標やY座標に加算する数値の事です。

任意の方向の移動量を知るには、sin関数とcos関数を使います。

X軸の移動量はcos関数に、Y軸の移動量はsin関数に、それぞれ角度（ラジアン単位）を引数にして受け渡す事で得られます。

※ラジアンと、弧度法の相互変換は第12章で解説しています。

6-2 目標へ向かって移動

目標へ移動する処理

目標への移動は、ゲーム、特にシューティングでは必須の処理でしょう。またシューティング以外でも、様々な所で使用され、非常に応用範囲の広い処理です。

目標の方向を調べる

さて、目標へ向かって移動する方法ですが、話自体は単純で目標への方向がわかればOKです。

方向がわかれば[6－1]で紹介したようにsin、cos関数を使用して目標方向への基本となる移動量を得る事が出来ます。

その目標への方向を調べるためには、atan2関数を使用します。

atan2関数に、出発点を起点とした、目標の座標を指定してやれば、角度（ラジアン単位）が得られます。

目標を狙うプログラム

ではプログラムです。サンプルは、画面中心から目標を狙ってボールが発射されるものです。

はじめに初期化処理が行なわれます。これは一定時間ごとに初期化されるもので、ボールの座標と目標の座標を初期化しています。

次に、座標から相手の方向を計算します。これは先ほどの説明通り、atan2関数を用いています。

ただ、座標をそのまま使うのではなく、スタート地点から見た、目標の座標を渡すことに注意してください。

あとは、得られた目標の方向をsin、cos関数に渡して、移動量を取得し、表示座標に加算してやります。

これで目標に向かって移動する事が出来ます。

LIST 6 ー2 目標に向かって移動

```
typedef struct{
    SPRITE          Ball;
    SPRITE          Target;                 //目標のスプライト
    int             Time;
} EX06_02_STRUCT ;

void init06_02(TCB* thisTCB)
{
    //使用するテクスチャの読み込み
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥0044.png",&g_pTex[0] );
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥0055.png",&g_pTex[1] );
}

void exec06_02(TCB* thisTCB)
{
#define START_X      SCREEN_WIDTH/2
#define START_Y      SCREEN_HEIGHT/2
#define TARGET_X     480
#define TARGET_Y     400
#define MOVE_SPEED   10.0

    EX06_02_STRUCT* work = (EX06_02_STRUCT*)thisTCB->Work;
    float direction;      //進行方向

    //初期化処理、一定時間ごとに初期化される
    if( work->Time-- == 0 )
    {
        work->Ball.X    = START_X;
        work->Ball.Y    = START_Y;
        work->Target.X = TARGET_X;
        work->Target.Y = TARGET_Y;

        work->Time = 30;
    }
    //座標から相手への方向を計算
    direction = atan2( TARGET_Y - START_Y, TARGET_X - START_X );
    //方向から、X、Yそれぞれの座標増分値を計算
```

```
    work->Ball.X += cos( direction ) * MOVE_SPEED;
    work->Ball.Y += sin( direction ) * MOVE_SPEED;

    SpriteDraw( &work->Ball, 1);
    SpriteDraw( &work->Target, 0);
}
```

なお、目標へ向かって移動する別の方法として、目標との座標を直接正規化する事で、目標への基本移動量を得る事も出来ます。

全部は紹介しませんが、仮想コードを示しておきますので、興味のある方は参考にして下さい。

```
//座標から正規化値を求める
idou_ryou = sqrtf( (pos.x * pos.x) + (pos.y * pos.y) );
//相対座標に正規化値を掛けて移動量とする、最後に表示座標に移動量を加算してやる
add_x = idou_ryou * pos.x;
add_y = idou_ryou * pos.y;
```

6-3 キャラクターを曲線的に移動

曲線的な移動処理

キャラクターを曲線的に移動させる処理を作成してみましょう。

通常、こういった処理はなかなか面倒な事が多いのですが、sin関数を使うと手軽に作成することが出来ます。

実際の処理

初期化

早速プログラムを見ていきましょう。サンプルは左右に動くボールをＺキーを押す事で上下に曲線的に移動させるものです。

はじめに初期化です。ここでは使用するデータの読み込みと、座標、及び左右の移動速度の初期化を行なっています。

メイン部分

次に、メイン部分です。まずボールを左右に移動させる処理を行ないます。この処理は一定時間ごとに、左右の移動方向を切り替える事で行なっています。

キー入力チェック

次にキー入力のチェックを行ない、キー入力中は曲線運動用の数値(sin関数に渡す数値)を加算し続けます。

この時加える数値によって、動きの速度が変わります。

そして移動処理です。Ｘ方向の移動は直線的な動きだけですので、加算のみを行ないます。

Ｙ方向の処理は移動の基本位置となる座標に、sin関数 から戻ってきた値を曲線の幅分だけ拡大して、加えてやります。

この時関数に渡す値は、先ほど計算した曲線運動用の数値です。

以上でsin関数を利用した曲線的な動きが可能になります。

sin関数は、ゲームでは非常に便利に使われる関数ですので、この例をはじめとして、是非使い方を覚えておいてください。

なお、曲線的な移動を行なうにはこの手法以外にも幾つかあり、代表的なものとしてベジェ等の

スプライン曲線があります。
　興味のある方や、この動きに満足できない方は、調べてみると良いでしょう。

LIST 6 －3－1 キャラを曲線的に移動

```
typedef struct{
    SPRITE          Ball;
    float           Time;
    float           AddX;
    float           SinNum;
} EX06_03_STRUCT ;

void init06_03(TCB* thisTCB)
{
#define MOVE_SPEED  2.0
    EX06_03_STRUCT* work = (EX06_03_STRUCT*)thisTCB->Work;
    //使用するテクスチャの読み込み
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥0055.png",&g_pTex[0] );

    //座標の初期化
    work->Ball.X    = SCREEN_WIDTH  / 3;
    //移動速度の初期化
    work->AddX      = -MOVE_SPEED;
}

void exec06_03(TCB* thisTCB)
{
#define CHANGE_TIME 100         //左右移動切り替え時間
#define SIN_WIDTH   50.0        //カーブの揺れ幅
#define SIN_SPEED   0.1         //振幅の速度

    EX06_03_STRUCT* work = (EX06_03_STRUCT*)thisTCB->Work;


    //一定時間ごとに左右移動を切り替える
    if( work->Time-- <= 0 )
    {
        //移動方向を反転させる
        work->AddX *= -1;
```

```
        //次の切り替え時間
        work->Time = CHANGE_TIME;
    }

    //Zキーを押している間は曲線用の値を加算
    if( g_InputBuff & KEY_Z ) work->SinNum += SIN_SPEED;

    //左右移動は直線的な運動のみ
    work->Ball.X += work->AddX;

    //上下移動はSin関数を用いて曲線状にする
    work->Ball.Y = SCREEN_HEIGHT / 2; //基本座標
    work->Ball.Y += sin( work->SinNum ) * SIN_WIDTH;

    SpriteDraw( &work->Ball, 0);
}
```

6-4 スクロールに合わせてキャラを動かす

相対座標

スクロールに合わせてキャラを動かす処理を作ってみましょう。

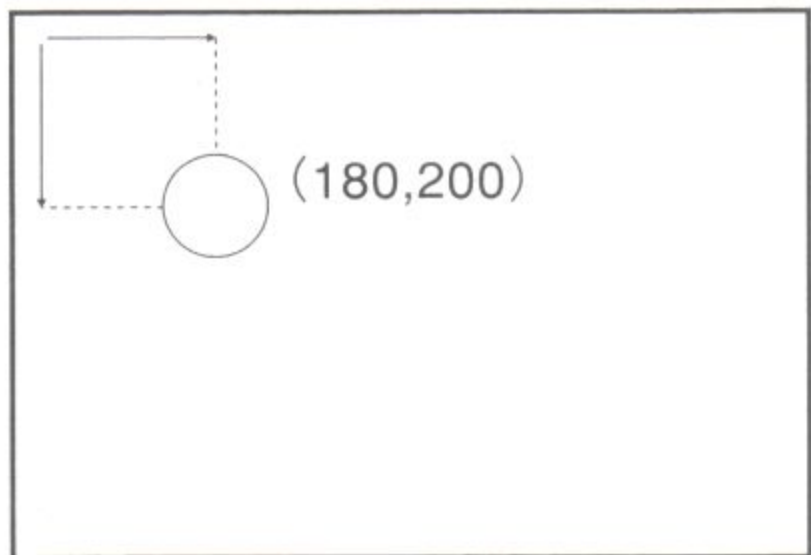
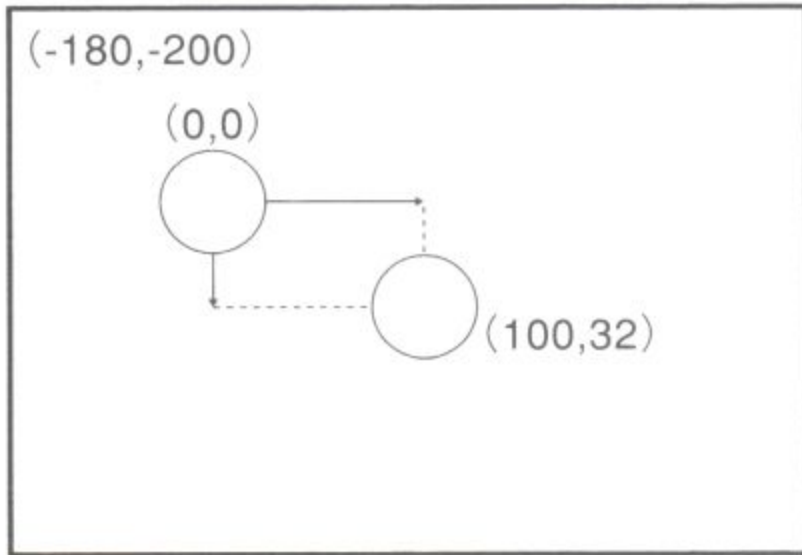
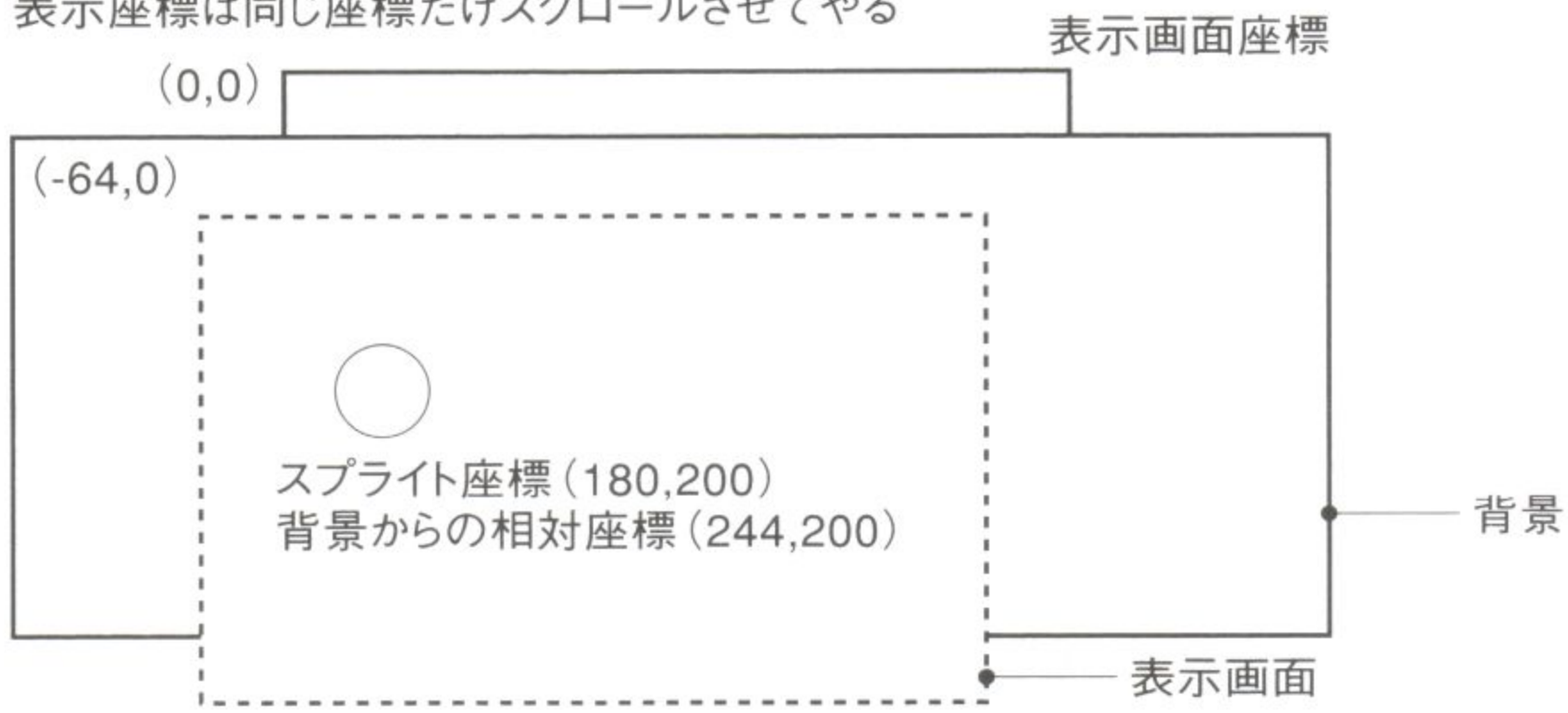
この処理はシューティングで背景と共に移動するキャラや、アクションゲームで背景に配置する物体等、スクロールするゲームにおいては、非常に良く使われる処理です。

では、この処理を行なうにはどのようにすれば良いのでしょうか？

それには、相対座標の概念を用います。

相対座標とは、ある特定の座標に対しての座標の事で、通常、画面に対して行なう座標管理を、特定の座標に対して行ないます。

この特定の座標をスクロール座標とし、それに対して表示座標を管理するようにしてやれば、この処理は実現できます。

図6－4－1 相対座標の概念図

スクロールにあわせて動かすプログラム

プログラム概要

ではプログラムを見ていきましょう。

サンプルは、Z、X2つのキーで左右にスクロールする背景と、それに合わせて移動が可能な、自機のプログラムです。

ここでは、シンプルに2つの座標のみを扱っています。すなわち、スクロール座標と、自機の座標です。

背景スクロール

最初に、背景のスクロールです。まず、キー入力を見て背景の座標を左右に移動させ、その後表示してやります。

この時、直接スクロールさせるのではなく、後から自機が座標を参照できるように、外部に座標を保持しておきます。

これは、構造体でScrollPosXとなっている変数ですが、左右にスクロールするだけですので、X座標だけ外部に保持しています。

自機の移動

次に、自機の移動処理です。方向キーをみて自機を入力されて方向に移動させます。

この時、移動する座標は、画面上での座標ではなく、スクロールに対する相対座標である事に注意してください。

自機の座標を算出

移動後、スクロール座標と自機の相対座標から、自機の表示座標を算出します。これは両方の座標を加算するだけで得られます。

最後は得られた表示座標に自機を表示して、処理は終了です。

LIST 6－4－1 スクロールに合わせてキャラを動かす

```
typedef struct{
    SPRITE          MyShip;
    BACK_GROUND     BG;
    int             ScrollPosX;
    int             MyShipPosX;
    int             MyShipPosY;
} EX06_04_STRUCT;
```

```
void init06_04(TCB* thisTCB)
{
    //使用するテクスチャの読み込み
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥0022.png",&g_pTex[0] );
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥0044.png",&g_pTex[1] );
}

void exec06_04(TCB* thisTCB)
{
#define MOVE_SPEED 8
#define SCROLL_SPEED 2
    EX06_04_STRUCT* work = (EX06_04_STRUCT*)thisTCB->Work;

    if( g_InputBuff & KEY_Z    )work->ScrollPosX -= SCROLL_SPEED;
    if( g_InputBuff & KEY_X    )work->ScrollPosX += SCROLL_SPEED;

    //背景をループさせる
    work->BG.X = work->ScrollPosX % 640;
    BGDraw(&work->BG,0);


    //キー入力による移動
    if( g_InputBuff & KEY_UP    ) work->MyShipPosY -= MOVE_SPEED;
    if( g_InputBuff & KEY_DOWN  ) work->MyShipPosY += MOVE_SPEED;
    if( g_InputBuff & KEY_RIGHT ) work->MyShipPosX += MOVE_SPEED;
    if( g_InputBuff & KEY_LEFT  ) work->MyShipPosX -= MOVE_SPEED;

    //スクロールに合わせる
    work->MyShip.X = work->ScrollPosX + work->MyShipPosX;
    work->MyShip.Y = work->MyShipPosY;

    SpriteDraw(&work->MyShip,1);

}
```

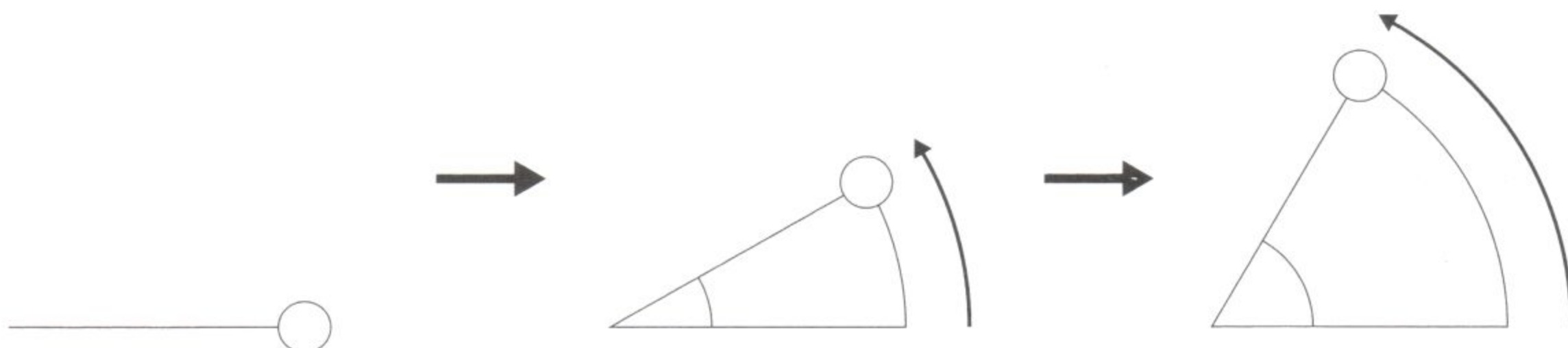
6-5 円運動をする

円運動を実現するには

円運動は直線運動と同様、ゲームでの基本となる動きです。

移動はもちろん、演出やエフェクト等にも使用されます。また、直線運動と違って円弧を描くため、曲線的な移動として用いられる事もあります。

さて円運動を行なうにはsin、cos関数を使用して円の角度に対応した座標を計算し、角度を少しずつずらしていく事で、実現できます。

図6－5－1 円運動の概念図

角度を増加させる事で円運動を行なう

円運動のプログラム

ではプログラムを見ていきます。サンプルは、画面の中心座標を元に、キャラが円運動を行なうものです。

円運動の半径を決める

まず最初に、中心点の座標と円運動を行なう円の半径を決めておきます。

この中心点が表示の中心座標にもなります。サンプルではマクロを用いていますが、もちろん変数でも構いません。

```
#define CIRCLE_CENTER_X 320
#define CIRCLE_CENTER_Y 240
#define CIRCLE_RADIUS   160
```

速度を決める

次に、円の角度を計算します。ここでは毎フレーム、角度の増加を行なっています。この増加の度合いで、円運動の速度が決まります。

座標を出す

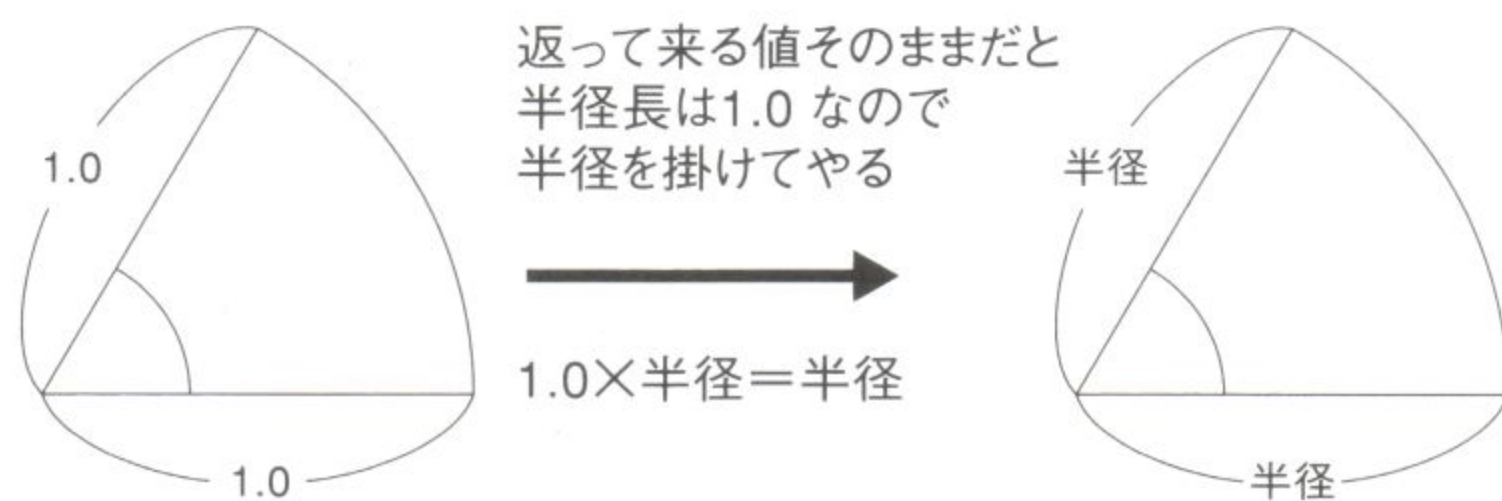
その後、角度を、sin、cos関数に引数として渡してやり、基本となる座標方向の基本移動量を取得します。

その基本移動量に半径をかけて座標を算出します。少し分かりにくいですが、得られた方向に座標を伸ばしてやるイメージをもつと分かりやすいかと思います。

ここで得られる座標は、座標に対する座標ですので、最後に中心座標を加算して最終的な座標を得ます。

これを毎フレーム行なえばOKです。

図6－5－2 円の座標算出の図

1.0
1.0
返って来る値そのままだと
半径長は1.0なので
半径を掛けてやる
1.0×半径＝半径
半径
半径

X＝cos（角度）×半径
Y＝sin（角度）×半径

角度に対するX,Y座標はcos,sinの各関数で行なう
ただし、返って来る値は1.0を基準としているので半径を掛けてやる

LIST 6－5－1 円運動

```
void init06_05(TCB* thisTCB)
{
    //使用するテクスチャの読み込み
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥0055.png",&g_pTex[0] );
}

void exec06_05(TCB* thisTCB)
{
#define CIRCLE_CENTER_X 320-16    //回転中心点X
```

```
#define CIRCLE_CENTER_Y 240-16    //回転中心点Y
#define CIRCLE_RADIUS   80        //回転半径
#define CIRCLE_SPEED    0.05      //回転速度

    SPRITE* pspr;
    pspr = (SPRITE*)thisTCB->Work;
    //回転速度
    pspr->Count += CIRCLE_SPEED;
    //円運動計算
    pspr->X = CIRCLE_CENTER_X + sin(pspr->Count) * CIRCLE_RADIUS;
    pspr->Y = CIRCLE_CENTER_Y + cos(pspr->Count) * CIRCLE_RADIUS;

    SpriteDraw(pspr,0);
}
```

6-6 楕円運動をする

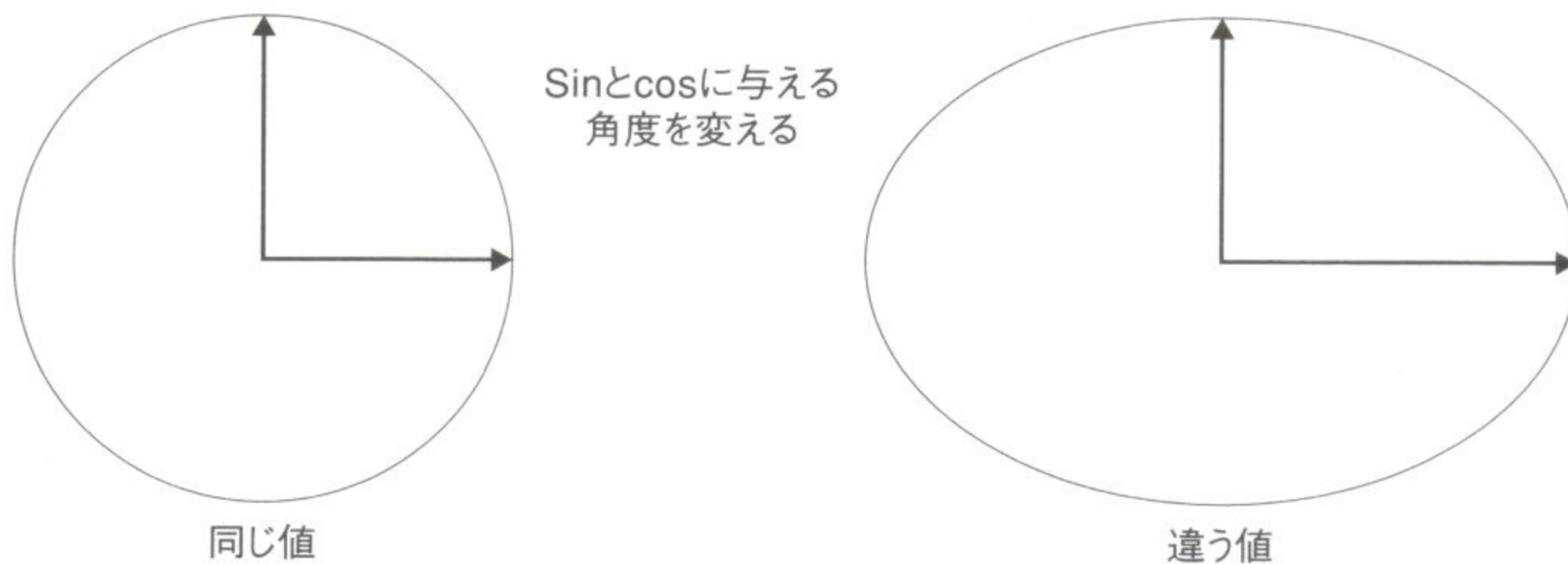
楽に楕円を表現する

円運動に引き続いて楕円運動を行なってみましょう。楕円運動も円運動と同様に、応用範囲の広い動きです。

ところが、実際に楕円運動を行なおうとすると、楕円の方程式を解かねばならず、結構厄介な事が分かります。

実は方程式を解かずとも、もっと手軽な方法があります。それは、[6－5]の円運動を基本の動きとして、円を歪ませてやるのです。

単純に円を歪ませるため、厳密には楕円とは呼べず、制御も少し難しいですが、ゲームで使う上では十分な動きです。

▲図6－6－1 円を歪ませて楕円を得る図

sin,cos関数に与える値を変更したり、ずらしてやる事で円を歪ませる

楕円の動きのプログラム

では、プログラムを見て行きましょう。

とは言うものの、基本的には円のプログラムとかなり似ていますので、説明するところはあまり多くありません。

ただ、動きが分かりやすいよう、表示するボールの数を増やすようにしています。

その為、メインとなる関数 exec06_06 は一定数に達成するまで、ボールを作成しつづけるように変更しています。

また同時に、楕円の歪みを決める「歪み値」を管理しており、毎フレーム歪み値を加算する事で楕円に動きを持たせています。

```
void exec06_06(TCB* thisTCB)
{
    EX06_06_STRUCT* work = (EX06_06_STRUCT*)thisTCB->Work;

    if(work->Time++ == 0x08)
    {//8フレーム毎にボールを作成
        work->Time = 0;
        if(work->BallCount++ < BALL_COUNT)
        {//一定数作成するまで繰り返す
            TaskMake(exec06_06_ball,0x1000);
        }
    }
    //歪み値
    EX06_06_gDist += 0.01;
}
```

さて、実際の楕円運動ですが、基本的には円運動とほぼ同じです。

違うポイントは、先ほど計算した「歪み値」を計算過程に組み込むところです。。

```
    //回転速度
    pspr->Count += CIRCLE_SPEED;

    //楕円運動計算
    pspr->X = CIRCLE_CENTER_X + sin(pspr->Count-EX06_06_gDist) *
CIRCLE_RADIUS;
    pspr->Y = CIRCLE_CENTER_Y + cos(pspr->Count+EX06_06_gDist) *
CIRCLE_RADIUS;
```

この計算過程により、歪ませ方は幾つかありますが、sin関数とcos関数に与える角度をずらしてやるのが効果的です。

ここでは、歪み値をsin、cos両関数にずらして与える事によって、楕円を実装しています。

この歪み値がどう影響を与えるかは、言葉では説明しにくい面があるため、実際に数値を変更しながら確認してもらったほうが理解がスムーズに行くかと思います。

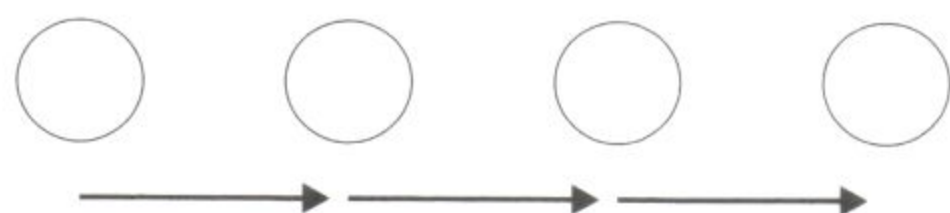
6-7 キャラに加減速をつけて移動

加減速運動

加減速運動とは、すなわち等加速度運動の事です。学校で習ったので覚えている方も多いでしょう。

ゲーム上では等速で移動する物体に対して、一定の割合で速度を加えてやる事（速度に加速値を加算する事）で実現します。

加速運動も減速運動も、加算する加速値が違うだけで処理はまったく同じです。

図6－9－1 等速運動と等加速度運動の比較図

等速度運動
動く速度は常に一定

加速度運動
移動速度が徐々に増えていく

加減速運動のプログラム

初期化

まずは初期化です。ここでは２つのタスクを生成して、加速、減速両方の動きを比較します。
次に作成したタスクのタスクワーク上に設けられた、Acc_Speedに加速値を設定しています。
数値が大きければ、加速の度合いも大きくなります。減速する場合はこの値を負にします。

LIST 6－7－1

```
typedef struct{
    SPRITE          sprt;
    float           Acc_Speed;
    float           Speed;
} EX06_07_STRUCT ;
void init06_07(TCB* thisTCB)
{
    //使用するテクスチャの読み込み
```

```
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥0055.png",&g_pTex[0] );
}

void exec06_07(TCB* thisTCB)
{
#define START_LINE      64          //開始ライン
#define ACC_SPEED       0.1         //速度
    TCB* child_tcb;
    EX06_07_STRUCT* work;

    child_tcb = TaskMake(exec06_08_acc,0x2000);
    work = (EX06_07_STRUCT*)child_tcb->Work;
    work->sprt.X= 400;
    work->sprt.Y= START_LINE;
    work->Acc_Speed = ACC_SPEED;

    child_tcb = TaskMake(exec06_08_acc,0x2000);
    work = (EX06_08_STRUCT*)child_tcb->Work;
    work->sprt.X= 240;
    work->sprt.Y= START_LINE;
    work->Acc_Speed = -ACC_SPEED;
    work->Speed     = 8.0;          //初速値


    TaskKill( thisTCB );
}
```

加減速処理

実際に加速処理を行なう部分です。

移動速度 Speed に対して毎フレーム、加速値 Acc_Speed を加算してやります。

ただしこのままだと、永遠に加速し続けてしまうので、一定ラインに到達したところで、速度、加速値を0に戻し、停止させています。

LIST 6 −7−2

```
void exec06_07_acc(TCB* thisTCB)
{
#define STOP_LINE       380       //停止ライン

```

```
    EX06_07_STRUCT* work = (EX06_08_STRUCT*)thisTCB->Work;

    work->Speed   += work->Acc_Speed;
    work->sprt.Y  += work->Speed;

    if( work->sprt.Y > STOP_LINE )
    {
        work->Acc_Speed   = 0.0;
        work->Speed       = 0.0;
    }

    SpriteDraw(&work->sprt,0);
}
```

6-8 誘導弾を撃つ 1

誘導弾とは

ゲーム、特にシューティングでは誘導弾は必ずといって良いほど出てきます。

またシューティングに限らなくとも、演出や移動など、誘導や追尾のアルゴリズムはリアルタイムのゲームでは必須といっても良いでしょう。

ここでは、誘導弾の種類をいくつか紹介します。

また誘導弾に絞って解説していますが、基本さえしっかり抑えておけば応用は難しくありません。

誘導弾のプログラム

初期化

まずは作成、初期化の処理です。

追尾をするには、当然ながら誘導弾と目標が必要です。最初に目標となるタスクと、誘導弾のタスクの２つを生成しています。

作成時のポイントとしては、目標となるタスクへのポインタを誘導弾のタスクワークに登録している所です。

こうする事で、誘導弾のタスクは常に目標を監視する事ができます。

LIST 6 －８－１ 誘導弾1

```
typedef struct{
    SPRITE          sprt;
    SPRITE*         Target;                    //目標のスプライト
} EX06_08_STRUCT ;

void init06_08(TCB* thisTCB)
{
    //使用するテクスチャの読み込み
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥0044.png",&g_pTex[0] );
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥0055.png",&g_pTex[1] );
}
```

```
void exec06_08(TCB* thisTCB)
{
    TCB*    tmp_tcb;
    SPRITE* target_sprt;
    EX06_08_STRUCT* tmp_work;

    //目標のスプライト
    tmp_tcb = TaskMake( exec06_08_target, 0x1000 );
    target_sprt = (SPRITE*)tmp_tcb->Work;
    target_sprt->X = SCREEN_WIDTH/2;
    target_sprt->Y = SCREEN_HEIGHT/2;

    //誘導弾の初期化と、目標スプライトを設定
    tmp_tcb = TaskMake( exec06_08_horming, 0x2000 );
    tmp_work = (EX06_08_STRUCT*)tmp_tcb->Work;
    tmp_work->Target = target_sprt;


    //生成後、処理終了
    TaskKill( thisTCB );
}
```

目標のスプライト処理

では実際に、処理を行なうタスクを見ていきましょう。
まず、目標となるスプライトを処理するタスクです。
スプライトは、入力キーの上下左右で8方向に移動します。
移動以外の処理はしていないので、特に問題はないでしょう。

LIST 6 −8−2

```
void exec06_08_target(TCB* thisTCB)
{
#define MOVE_SPEED  8.0
    SPRITE* work = (SPRITE*)thisTCB->Work;

    //キー入力による移動
    if( g_InputBuff & KEY_UP    ) work->Y -= MOVE_SPEED;
    if( g_InputBuff & KEY_DOWN  ) work->Y += MOVE_SPEED;
```

```
    if( g_InputBuff & KEY_RIGHT ) work->X += MOVE_SPEED;
    if( g_InputBuff & KEY_LEFT  ) work->X -= MOVE_SPEED;

    SpriteDraw( work, 0);
}
```

誘導処理

次に実際に追尾を行なうタスクです。

まず、追尾目標となる座標を取得します。これは、タスクの生成時に格納された目標のタスクワークから取得します。

```
    float targetX = work->Target->X + 32;
    float targetY = work->Target->Y + 32;
```

次に、自己座標と目標の座標から、目標への方向を取得します※。

```
    direction= atan2(  targetY - my_Y , targetX - my_X);
```

後は目標への方向から、進行方向への基本移動量を割り出します。
その後移動速度を掛けて、実際の移動量を算出します。
後は、その移動量を現在の座標に加算してやればOKです。

LIST 6 －8－3

```
void exec06_08_horming(TCB* thisTCB)
{
#define MOVE_SPEED  2.0
    EX06_08_STRUCT* work = (EX06_08_STRUCT*)thisTCB->Work;
    float my_X    = work->sprt.X;
    float my_Y    = work->sprt.Y;
    //目標の座標を取得
    float targetX = work->Target->X + 32;
    float targetY = work->Target->Y + 32;

    float direction;

```

※詳細は[6－2]を参照。

```
    //目標の方向を取得
    direction= atan2(  targetY - my_Y , targetX - my_X);

    //方向への移動値を取得
    work->sprt.X += cos(direction)*MOVE_SPEED;
    work->sprt.Y += sin(direction)*MOVE_SPEED;

    SpriteDraw(&work->sprt,1);
}
```

6-9 誘導弾を撃つ 2

加速しながら追いかける誘導弾

2つ目の誘導弾は加速しながら目標を追尾する動きです。

挙動のイメージとしては、常に目標に向かって落下し続けている形になるでしょうか。

基本的な初期化、タスク作成処理は[6－8]と変わっていません。

スプライトの動きも同様です。

このあたりの処理の詳細は[6－8]を参照してください。ここでは追尾処理をメインに解説します。

加速する処理

まず、通常の追尾と同様に目標の座標を取得します。

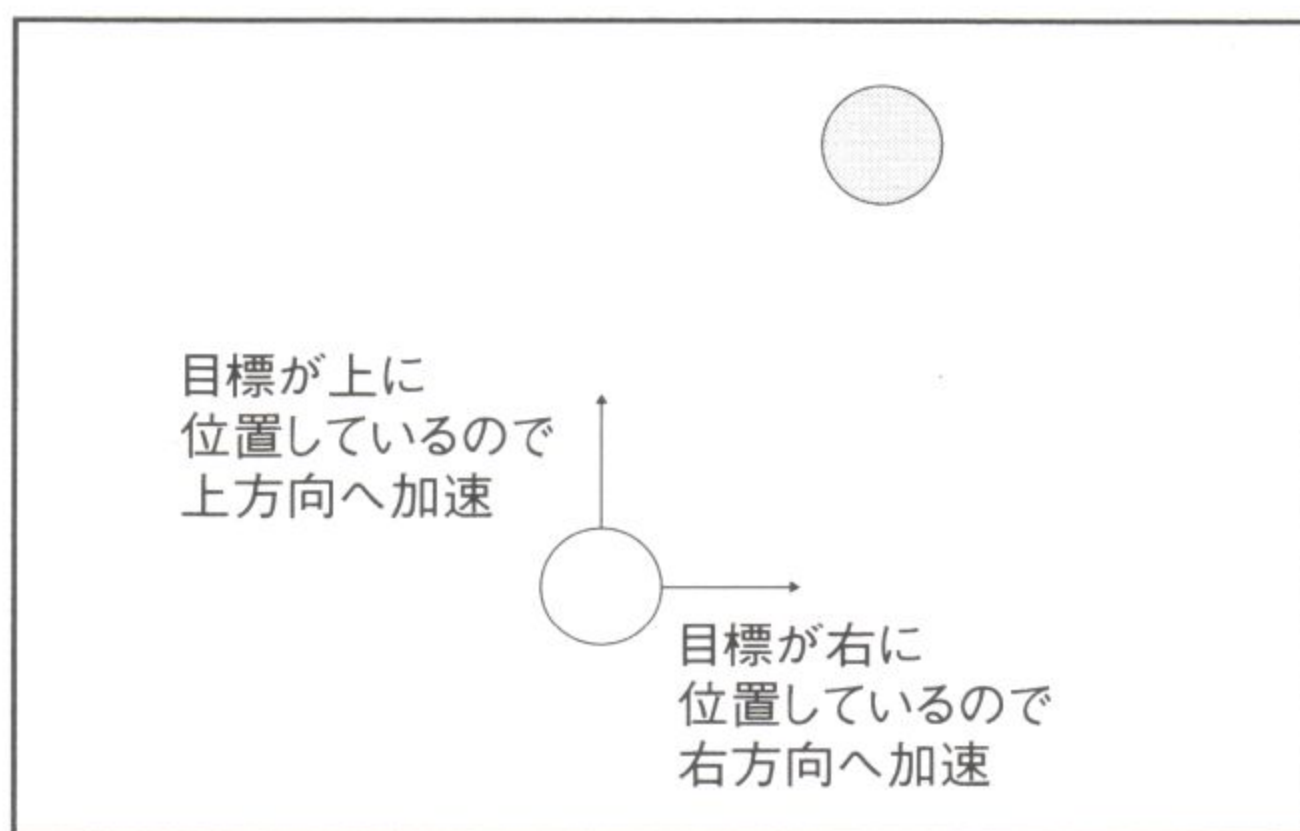
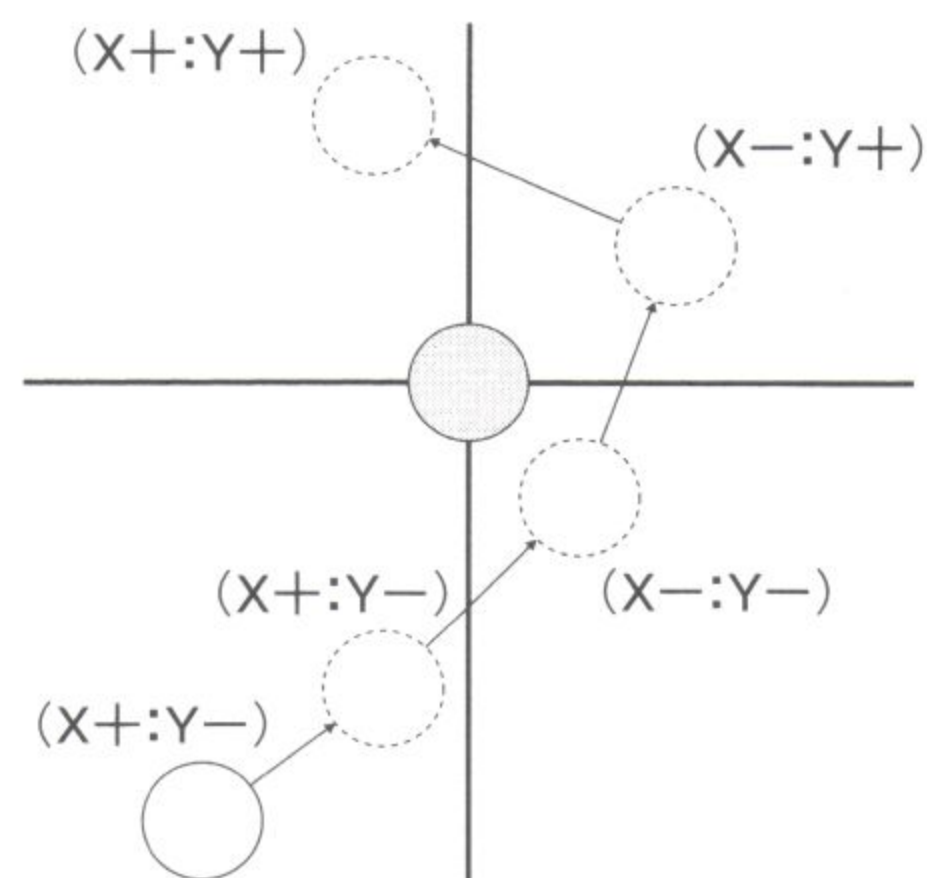
その後、目標との相対位置を元に、X軸Y軸それぞれに加速する方向を決定します。

具体的には、離れている方向の軸に向かって加速を行ないます。

最後に算出された方向を元に等加速度運動を行ないます※。

なお、ここでは加速の方向は4方向ですが、精度を重視するケースでない限りは十分な動きです。

精度を重視したい場合は、目標への方向を取得した後、基本移動量に対して等加速度運動を行なうと良いでしょう。

図6－9－1 目標方向への加速のイメージ図

目標への加速度運動
目標の位置に応じて、X,Y の各方向を加速してやる

目標への位置と加速の方向

※[6－8]も参照

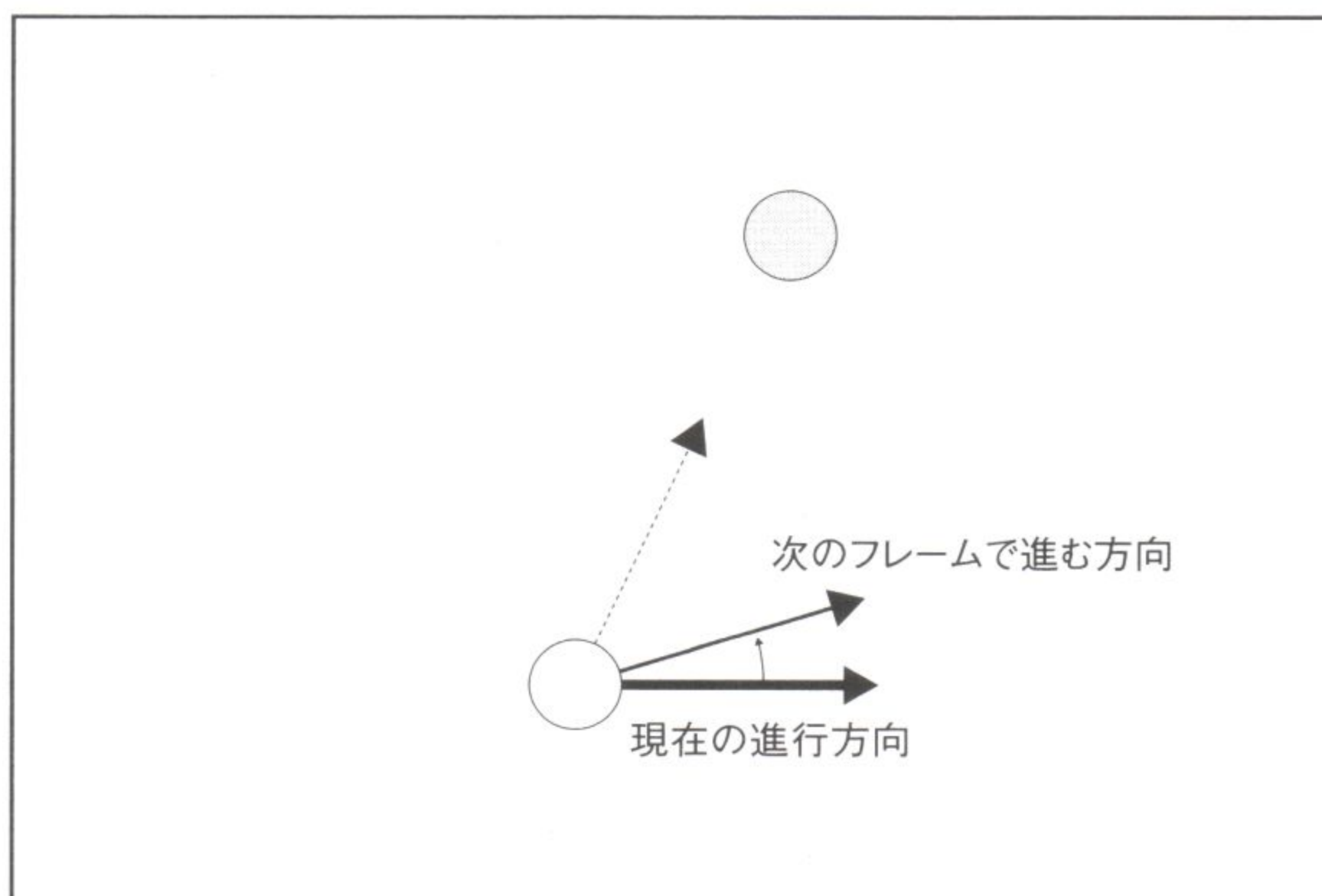
LIST 6 －9－2 誘導弾2

```
typedef struct{
    SPRITE          sprt;
    SPRITE*         Target;                 //目標のスプライト
    float           SpeedX;                 //速度
    float           SpeedY;
} EX06_09_STRUCT ;


void exec06_09_horming(TCB* thisTCB)
{
#define ACC_SPEED  1.0
    EX06_09_STRUCT* work = (EX06_09_STRUCT*)thisTCB->Work;
    float my_X    = work->sprt.X;
    float my_Y    = work->sprt.Y;
    //目標の座標を取得
    float targetX = work->Target->X + 32;
    float targetY = work->Target->Y + 32;

    float acc_speedX;
    float acc_speedY;
    float direction;

    //目標との相対位置を元に、加速方向を取得
    if( targetX - my_X >= 0) acc_speedX =  ACC_SPEED;
    if( targetX - my_X <  0) acc_speedX = -ACC_SPEED;
    if( targetY - my_Y >= 0) acc_speedY =  ACC_SPEED;
    if( targetY - my_Y <  0) acc_speedY = -ACC_SPEED;


    //加速値を元に速度を算出
    work->SpeedX += acc_speedX;
    work->SpeedY += acc_speedY;

    work->sprt.X += work->SpeedX;
    work->sprt.Y += work->SpeedY;

    SpriteDraw(&work->sprt,1);
}
```

6-10 誘導弾を撃つ 3

旋回して追いかける弾

3つ目の誘導弾はを回り込むように目標を追尾する動きです。
進行方向をすこしづつ目標に向かわせる形の誘導弾です。
実際のゲームでは誘導ミサイルなどに使われるケースが多いようです。

図6 ー10ー1 旋回して追いかける誘導弾のイメージ図

進行方向を目標の方向へ少しづつ向けてやる

旋回する動きのプログラミング

基本的な初期化、タスク作成処理は[6ー9]と変わっていません。スプライトの処理も同一です。
このあたりの処理の詳細は[6ー9]を参照してください。ここでは追尾処理に絞って解説します。

目標方向の取得と修正

まず、目標の座標から目標への方向を取得します。

```
direction= atan2( targetY - my_Y , targetX - my_X );
```

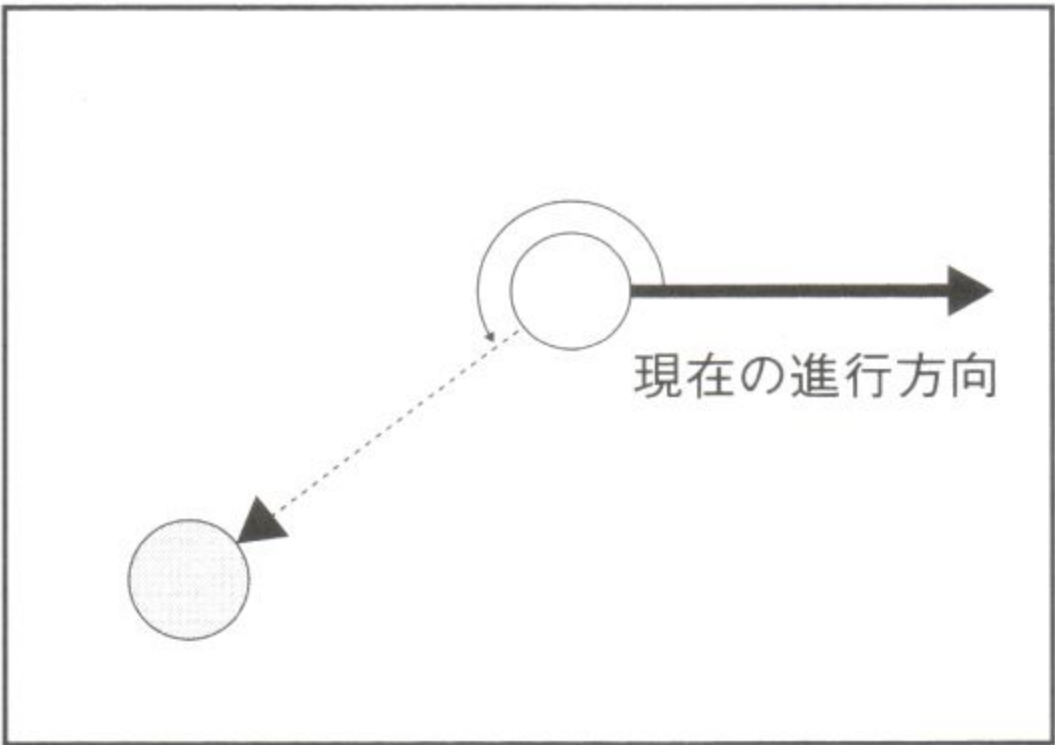
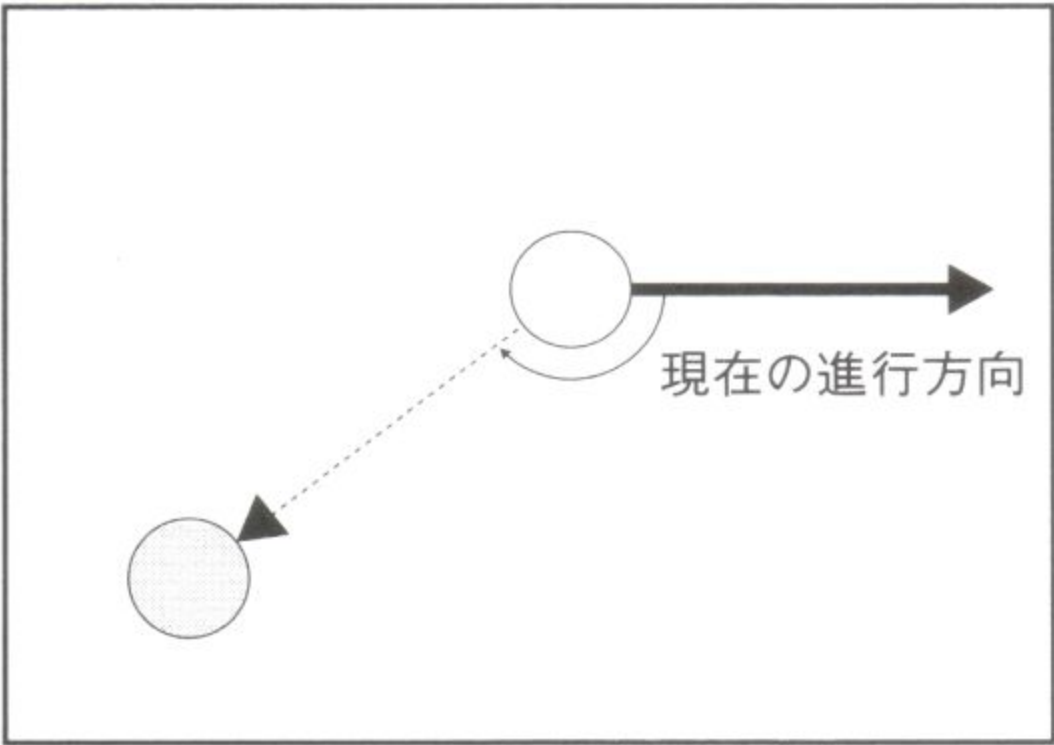
次に現在向いている方向と、目標との方向の差が180度を越えていた場合、その差を180度以下に丸め込みます。

これは正負の両方で行ないます。

こうしておかないと、180度以上方向差があった場合、遠回りの方向転換を行なってしまう場合があります。

LIST 6 －10－1

```
//方向を±PIの範囲内に丸める
    if( work->Direction - direction >  M_PI) work->Direction -= M_PI * 2;
    if( work->Direction - direction < -M_PI) work->Direction += M_PI * 2;
```

図6 －10－2　180度以上の角度差を丸め込む解説図

180度を超える角度は遠回りになるので　　近い方向になるように丸め込んでやる

旋回速度を加える

次に、現在向いている方向と目標の方向を比較し、方向が近づくように旋回速度を加えてやります。

旋回できる速度はマクロROT_SPEEDで定義しています。

最終的に算出された進行方向を元に移動量を決定します。最後に移動量を加算すれば完成です。

LIST 6 －10－2

```
typedef struct{
    SPRITE          sprt;
    SPRITE*         Target;               //目標のスプライト
    float           Direction;            //方向
} EX06_10_STRUCT ;

void exec06_10_horming(TCB* thisTCB)
```

```
{
#define ROT_SPEED  0.05     //旋回速度
#define MOVE_SPEED  4.0
    EX06_10_STRUCT* work = (EX06_10_STRUCT*)thisTCB->Work;
    float my_X     = work->sprt.X;
    float my_Y     = work->sprt.Y;
    //目標の座標を取得
    float targetX = work->Target->X + 32;
    float targetY = work->Target->Y + 32;

    float direction;


    //目標の方向を取得
    direction= atan2( targetY - my_Y , targetX - my_X );

    //方向を±PIの範囲内に丸める
    if( work->Direction - direction >  M_PI) work->Direction -= M_PI * 2;
    if( work->Direction - direction < -M_PI) work->Direction += M_PI * 2;

    //左回り
    if( work->Direction - direction < 0 ) work->Direction += ROT_SPEED;
    //右回り
    if( work->Direction - direction > 0 ) work->Direction -= ROT_SPEED;


    //進行方向への移動値を取得
    work->sprt.X += cos(work->Direction)*MOVE_SPEED;
    work->sprt.Y += sin(work->Direction)*MOVE_SPEED;

    SpriteDraw(&work->sprt,1);
}
```

6-11 移動用データを使ってキャラを動かす

データによる移動

データに合わせて、キャラクターを動かす処理を作成してみましょう。

この処理は、非常に多く使われる処理です。移動パターンは固定になりますが、キャラクターに思い通りに動きをつけたいときには有効です。

また、データを背景にあわせてやる事で、地形にあわせて移動するように見せる事も出来ます。

具体的な方法ですが、まず移動の元となる座標データを、移動するポイント分だけ用意します。このポイント同士を結ぶように移動してやれば、思い通りの座標を行き来する事が出来ます。

この方法であれば、データを増やしてやる事で、いくらでも細かい動きを行なう事が可能です。

実際のプログラム

ではプログラムを見ていきます。サンプルは、画面上方からボールが左右に蛇行しながら降りてくるものです。

まず、移動する座標のデータpoint_dataを用意します。これは単純な2次元配列で、X,Y座標が移動するポイントの個数分だけ並んでいます。

この、データの最後は特殊な数字(－999)で判断しています。

座標データの取得と移動

次に実際の移動方法です。

まず、移動の終了をチェックする為、時間のチェックを行ないます。

もし移動が終了していたら、次の座標データを読み込み、移動の設定を行います。

移動用データの設定

移動の際に必要となる、X,Y座標に加算する移動速度は、次のポイントへの方向と、速度から算出されます。

この時同時に、目標への距離も取得し、何フレームで到達するかの時間も計算しています。

取得した時間は、移動終了の為のタイマーとして設定されます。

これをデータが終了するまで繰り返せば、処理は終了です。

LIST 6 －11－1 データに沿ってキャラを動かす

```
#define MOVE_SPEED 8.0
```

```
#define END_DATA   -999.0
typedef struct{
    SPRITE      Sprt;
    int         Time;
    int         Point;
    //目標への移動速度
    float     AddX;
    float     AddY;
} EX06_11_STRUCT;

void init06_11(TCB* thisTCB)
{
    //使用するテクスチャの読み込み
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥0055.png",&g_pTex[0] );
}

void exec06_11(TCB* thisTCB)
{
    EX06_11_STRUCT* work = (EX06_11_STRUCT*)thisTCB->Work;
    float direction;
    float distance;
    float posX;
    float posY;
//移動先データ
float point_data[][2] =
{//   X      Y
    { 320.0,   0.0,},  //0 初期座標
    { 320.0,  96.0,},  //1
    { 480.0,  96.0,},  //2
    { 448.0, 256.0,},  //3
    {  96.0, 256.0,},  //4
    {  96.0, 320.0,},  //5
    { 560.0, 400.0,},  //6
    { 560.0, 480.0,},  //7
    {END_DATA,END_DATA,},  //終了
};

    //初期化時、または一定時間で目標を更新
    if(work->Time-- <= 0)
```

```
	{
		//次の目標をチェック
		if( point_data[work->Point+1][0] == END_DATA)
		{//データが終了していたら、データを初めに戻す
		 //(サンプルの場合 通常はそのまま終了処理)
			work->Point = 0;
		}

		//移動開始座標を指定
		//(移動終了時の誤差修正の意もある)
		work->Sprt.X = point_data[work->Point][0];
		work->Sprt.Y = point_data[work->Point][1];

		//目標からの相対座標を計算
		posX = point_data[work->Point+1][0] - work->Sprt.X;
		posY = point_data[work->Point+1][1] - work->Sprt.Y;

		//距離を計測し、速度から目標への所要時間を割り出す
		distance = sqrtf( (posX * posX) + (posY * posY) );
		work->Time = distance / MOVE_SPEED;

		//目標の方向計算
		direction = atan2( posY, posX);

		//方向から移動速度を設定
		work->AddX = cos( direction ) * MOVE_SPEED;
		work->AddY = sin( direction ) * MOVE_SPEED;

		//次のデータ
		work->Point++;
	}

	//移動処理
	work->Sprt.X += work->AddX;
	work->Sprt.Y += work->AddY;

	//ボール表示
	SpriteDraw( &work->Sprt, 0);
}
```

6-12 振り子の様な動き

振り子の動きを再現する

振り子の様な動きを再現してみましょう。

この様な動きはゲーム上で使う機会はあまりありませんが、周期的な動きのため、演出やボスなどの特殊な動きとして使われる事があります。

さて、振り子の動きを再現するには、幾つか方法があります。

もちろん、まともに計算しても良いのですが、ここではもっと手軽に「それっぽく」見える方法を紹介します。

具体的にどうするかと言うと、[6－5]で紹介した、円の動きを利用します。

振り子運動は実質的に円運動の一部でもあるため、移動速度を周期的に増減させれば、振り子のような動きにすることが出来ます。

振り子の動きのプログラミング

では実際にプログラムを見てみましょう。

プログラムは非常にシンプルで、振り子の移動速度となる角度値を計算し、その値を元にして運動計算を行なっています。

円運動と違う点

基本は円運動とほぼ同じですが、違う点として、計算した角度をそのまま角度値として利用するのではなく、一旦sin関数を通してから使用している点が挙げられます。

実はここがポイントで、sin関数は、周期関数であるため、引数となる角度を加算し続けるだけで、周期的な動きを実現できます。

また、周期の折り返し点、すなわち振り子の両端に近づくにしたがって、数値が減少するため、加減速がついているように見えるのです。

もっとも、先に説明したように、これは実際の振り子の動きそのものではありません。

より本物らしくするのであれば、等加速度運動を周期関数にして適用する等、少し面倒な手続きが必要になります。

LIST 6 －12－1 振り子のような動き

```
void init06_12(TCB* thisTCB)
{
    //使用するテクスチャの読み込み
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥0055.png",&g_pTex[0] );
}

void exec06_12(TCB* thisTCB)
{
#define PENDULUM_CENTER_X 320-16    //中心点X
#define PENDULUM_CENTER_Y 240-16    //中心点Y
#define PENDULUM_RADIUS   80        //振り子の半径
#define PENDULUM_SPEED    0.15      //往復速度
#define PENDULUM_RADIUS2  1.0       //振り子の往復の幅

    SPRITE* pspr;
    pspr = (SPRITE*)thisTCB->Work;
    //振り子の速度
    pspr->Count += PENDULUM_SPEED;

    //運動計算
    pspr->X = PENDULUM_CENTER_X + sin( sin( pspr->Count ) * PENDULUM_RADIUS2
) * PENDULUM_RADIUS;
    pspr->Y = PENDULUM_CENTER_Y + cos( sin( pspr->Count ) * PENDULUM_RADIUS2
) * PENDULUM_RADIUS;

    SpriteDraw(pspr,0);
}
```

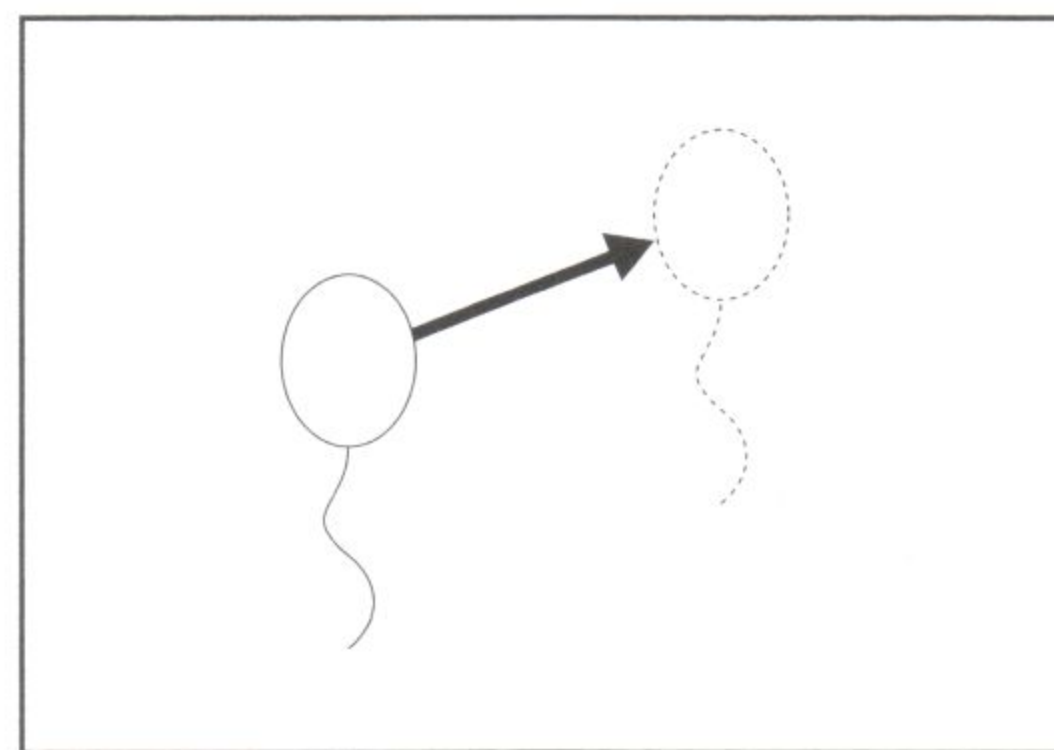
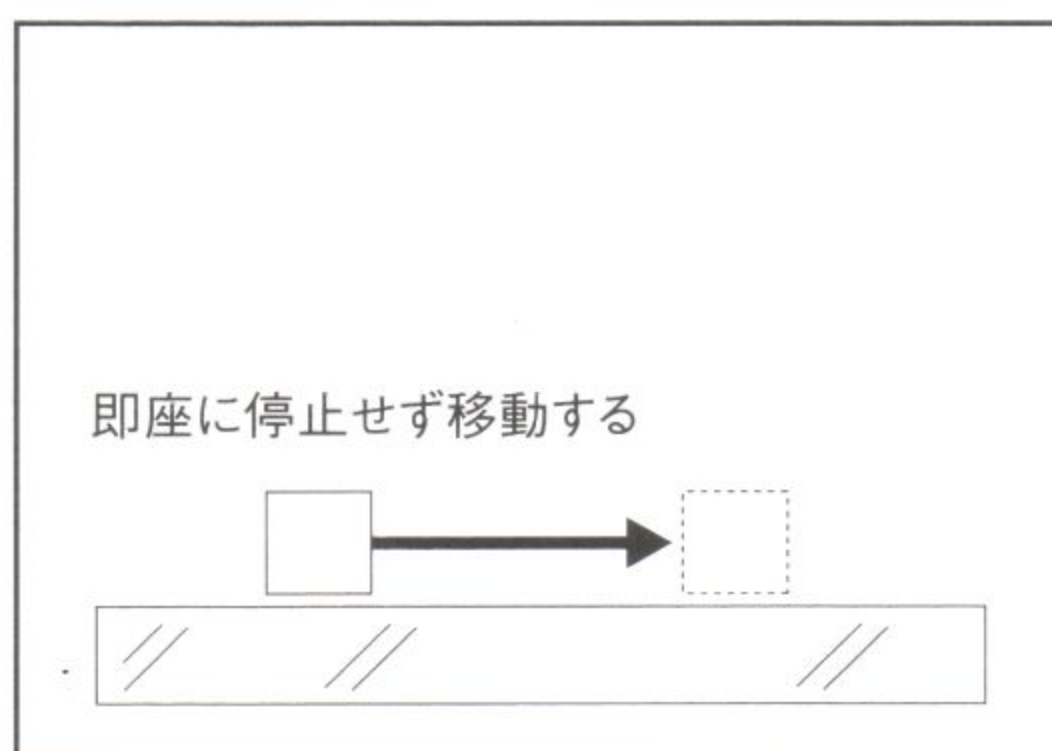
6-13 移動に慣性をつける

慣性をつける処理の概要

慣性の処理はアクションゲームで比較的良く見られる処理で、例えば走るのをやめて停止する時や、滑る床の表現をする時などに使われます。

見た目にもリアルさが増えますが、操作に独特のクセがあるため、アクション以外でも使用される事があり、ゲーム性を増すのにも一役買っています。

ここでは8方向に慣性をつけて動くプログラムを作ってみましょう。

図6－13－1 移動に慣性がつくアクションの例

氷の床や、風船、ビリヤードのボール等、操作を止めた場合でも、移動が行われる事がある為、ゲーム性や表現力が増す

処理の概要ですが、キー入力に対応して各方向に加速の処理を行ないます。

ただしこのままだと、移動し続けてしまうので、キー入力のしていない時は減速をしてやる必要があります。

この加減速のON/OFFで慣性のついた動きの表現を行ないます。

慣性処理のプログラミング

実際のプログラムでは以下のようになります。

まずはじめに、現在移動中かどうかを確認します。

もし移動中であれば、減速を行なう必要があるので、変数に減速用の値を代入します。

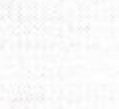
直接減速を行なっても良いのですが、加速値と減速値は同じ変数を使うため、キー入力があった時に変数を書き換えるだけで加速の処理を行なう事が出来ます。

次に加速処理です。キー入力を見ておき、反応があれば、減速と同様に数値を代入するだけです。

最後に加減速の計算処理を行ない、表示を行なってやり処理は終了です。

LIST 6 －13－1 移動に慣性をつける

```
typedef struct{
    SPRITE          Sprt;
    float           SpeedX;           //X方向への加速値
    float           SpeedY;           //Y方向への加速値
} EX06_13_STRUCT ;


void init06_13(TCB* thisTCB)
{
    EX06_13_STRUCT* work = (EX06_13_STRUCT*)thisTCB->Work;

    //使用するテクスチャの読み込み
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥0044.png",&g_pTex[0] );

    //自機のスプライト
    work->Sprt.X = SCREEN_WIDTH/2;
    work->Sprt.Y = SCREEN_HEIGHT/2;
}

void exec06_13(TCB* thisTCB)
{
#define ACC_SPEED  1.0
    EX06_13_STRUCT* work = (EX06_13_STRUCT*)thisTCB->Work;

    float acc_speedX = 0;
    float acc_speedY = 0;

    //もし、移動していたら減速を行う
    if(work->SpeedX > 0 ) acc_speedX = -ACC_SPEED;
    if(work->SpeedX < 0 ) acc_speedX =  ACC_SPEED;

    if(work->SpeedY > 0 ) acc_speedY = -ACC_SPEED;
    if(work->SpeedY < 0 ) acc_speedY =  ACC_SPEED;
```

```
    //キー入力により加速を行う
    if( g_InputBuff & KEY_UP    ) acc_speedY = -ACC_SPEED;
    if( g_InputBuff & KEY_DOWN  ) acc_speedY =  ACC_SPEED;

    if( g_InputBuff & KEY_LEFT  ) acc_speedX = -ACC_SPEED;
    if( g_InputBuff & KEY_RIGHT ) acc_speedX =  ACC_SPEED;

    work->SpeedX += acc_speedX;
    work->SpeedY += acc_speedY;
    work->Sprt.X += work->SpeedX;
    work->Sprt.Y += work->SpeedY;


    SpriteDraw( &work->Sprt, 0);
}
```

6-14 オプションの動き 1

キャラの周囲を旋回する

自機の動きに合わせて移動する、いわゆる“オプション”の動きを再現してみましょう。

シューティングは元より、格闘ゲームやアクションゲームでも使用されるため、応用はかなり広いと思われます。

オプションの動きは色々ありますが、ここでは代表的な動きを3種類ほどピックアップします。

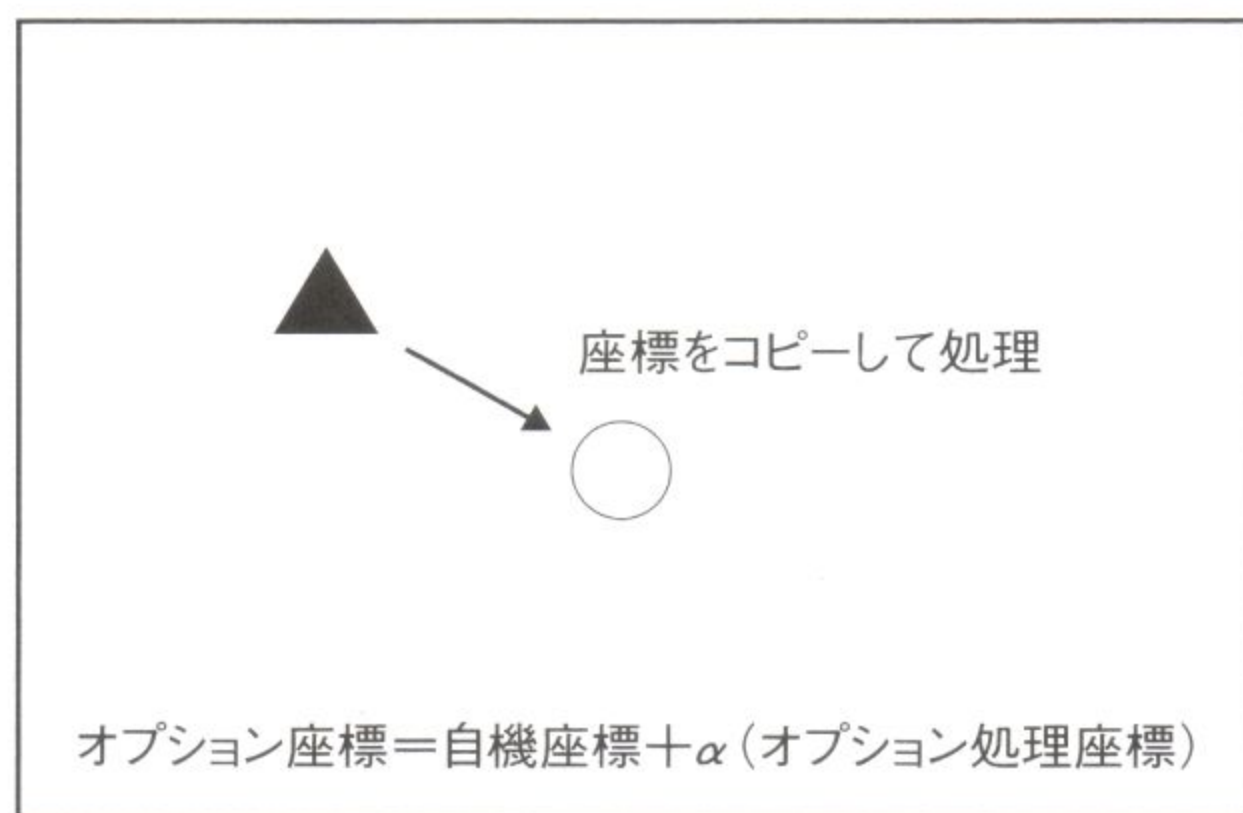
最初に、自機を中心に回転移動を行なうオプションを作成してみましょう。

基本となる考え方は、自機タスクとそれにくっつくオプションタスクを用意します。

オプションタスクは、常に自機の座標を見ており、その座標に合わせて回転移動を行ないます。

図6－14－1 自機座標を参照するオプションタスクの概念図

座標をコピーして処理

オプション座標＝自機座標＋α（オプション処理座標）

常に自機の座標を参照しその座標に対して処理を行なう

自機の周りに回すプログラム

初期化

プログラムを見ていきましょう。

最初に実行されるタスクは初期化処理用のタスクです。

初期化処理では、まず最初に自機の移動タスクを作成します。

この時自機タスクのワークへのポインタを記録しておくのを忘れないでください。

オプションタスク作成

次に、オプションタスクを作成します。

タスクの作成後は、オプションの個体を判別するためのIDと、先ほど記録した自機タスクのワークへのポインタを引き渡しておきます。

特にIDはオプションの個別認識をするために、必須の情報です。

ここでは4つのオプションタスクを作成しています。

初期化処理終了後、初期化タスクは自己消滅します。

LIST 6 －14－1

```
typedef struct{
    SPRITE          sprt;
    //自機情報
    SPRITE*         Myship;
    int             OptionID;
    //回転
    float           Rot;
} EX06_14_STRUCT ;

void init06_14(TCB* thisTCB)
{
    //使用するテクスチャの読み込み
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥0044.png",&g_pTex[0] );
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥0055.png",&g_pTex[1] );
}

void exec06_14(TCB* thisTCB)
{
    TCB*     tmp_tcb;
    SPRITE* my_ship_sprt;
    EX06_14_STRUCT* tmp_work;
    int loop;


    //自機のスプライト
    tmp_tcb = TaskMake( exec06_14_my_ship, 0x1000 );
```

```
        my_ship_sprt = (SPRITE*)tmp_tcb->Work;
        my_ship_sprt->X = SCREEN_WIDTH/2;
        my_ship_sprt->Y = SCREEN_HEIGHT/2;

        //オプションを4つ生成、自機ワーク情報を設定
        for( loop = 0; loop < 4; loop++)
        {
            tmp_tcb = TaskMake( exec06_14_option, 0x2000 );
            tmp_work = (EX06_14_STRUCT*)tmp_tcb->Work;
            tmp_work->Myship = my_ship_sprt;
            tmp_work->OptionID = loop;
        }
        //生成後、処理終了
        TaskKill( thisTCB );
}
```

自機タスク

次に個別のタスクを見ていきます。

自機タスクは自機の移動を管理します。

行なっている処理は入力キーに合わせて、自機を8方向に動かす事と、自機の表示です。

オプションタスク

さて、肝心のオプションタスクです。

最初に、自機の座標を取得します。

自機の座標情報は、タスク作成時に引き渡された、自機タスクへのワークへのポインタを使用して取得します。

この際、取得するワークと、引用するポインタの構造体を合わせなければなりませんので、注意が必要です。

ここでは、自機のタスクは、SPRITE構造体で座標を管理しており、ポインタもそれにあわせたものになります。

表示

最後に、取得した自機の座標と回転する座標を加算して座標を計算します。

その際、オプションIDを用いて、回転する位置を個別に管理してやります。

サンプルではIDに合わせて、回転位置を90度づつずらしています。

最後にオプションを座標に合わせて表示してやれば終了です。

LIST 6 －14－2

```
void exec06_14_option(TCB* thisTCB)
{
#define ROT_SPEED     0.05           //回転速度
#define ROT_RADIUS    96             //回転半径
//各オプションの回転間隔
float option_table[] = { 0.0, M_PI * 0.5, M_PI, M_PI * 1.5,};

    EX06_14_STRUCT* work = (EX06_14_STRUCT*)thisTCB->Work;
    //自機の座標を取得
    float my_shipX = work->Myship->X + 16;
    float my_shipY = work->Myship->Y + 16;

    work->Rot += ROT_SPEED;
    //精度低下を防ぐために丸める
    if( work->Rot >  M_PI) work->Rot -= M_PI * 2;

    work->sprt.X = my_shipX + cos(work->Rot + option_table[ work->OptionID ]
) * ROT_RADIUS;
    work->sprt.Y = my_shipY + sin(work->Rot + option_table[ work->OptionID ]
) * ROT_RADIUS;

    SpriteDraw(&work->sprt,1);

}

void exec06_14_my_ship(TCB* thisTCB)
{
#define MOVE_SPEED  8.0
    SPRITE* work = (SPRITE*)thisTCB->Work;

    //キー入力による移動
    if( g_InputBuff & KEY_UP     ) work->Y -= MOVE_SPEED;
    if( g_InputBuff & KEY_DOWN   ) work->Y += MOVE_SPEED;
    if( g_InputBuff & KEY_RIGHT  ) work->X += MOVE_SPEED;
    if( g_InputBuff & KEY_LEFT   ) work->X -= MOVE_SPEED;

    SpriteDraw( work, 0);
}
```

6-15 オプションの動き 2

常に自機を追いかける

２つ目のオプションは常に自機を追いかける動きを作成してみます。

このオプションの動きですが、自機の動きを常に記録しておき、記録した座標を各オプションが再生する事で実現できます。

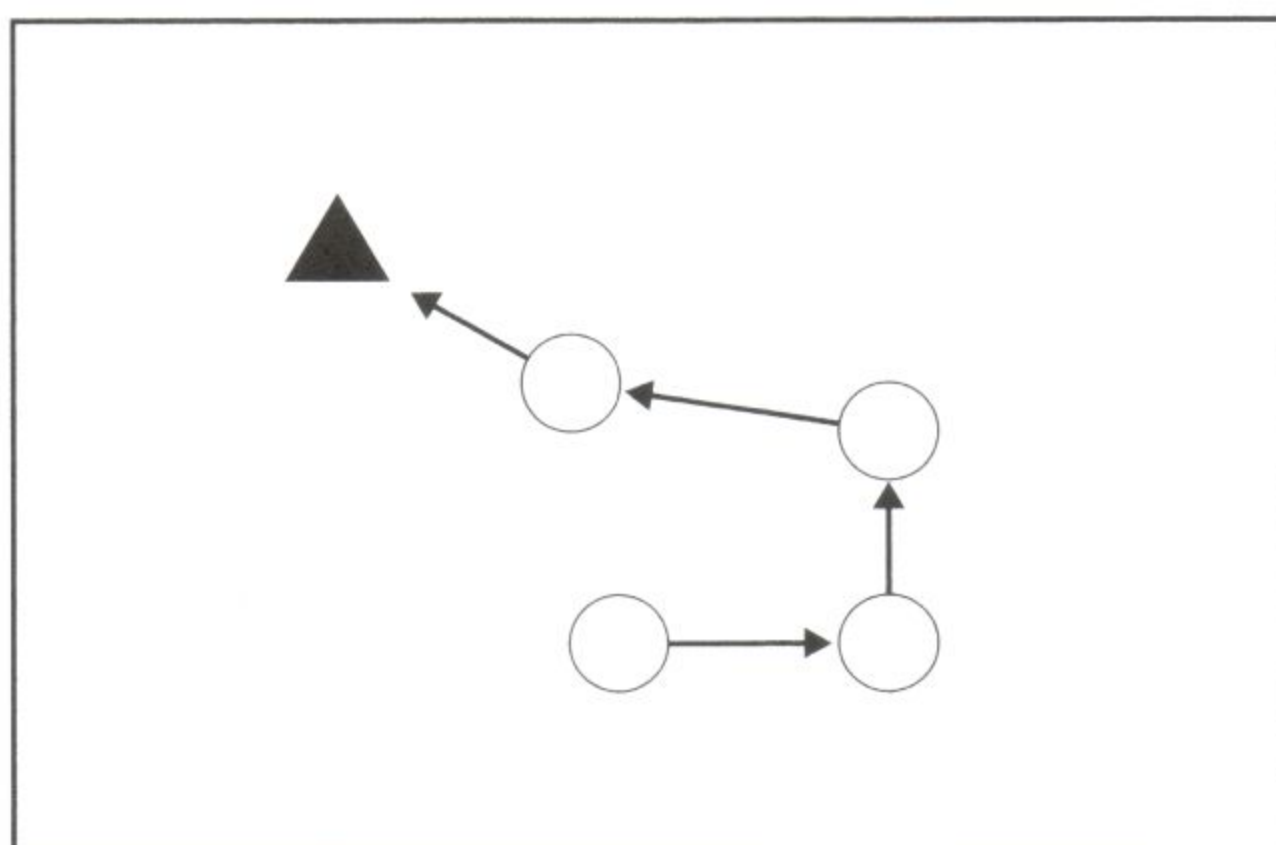
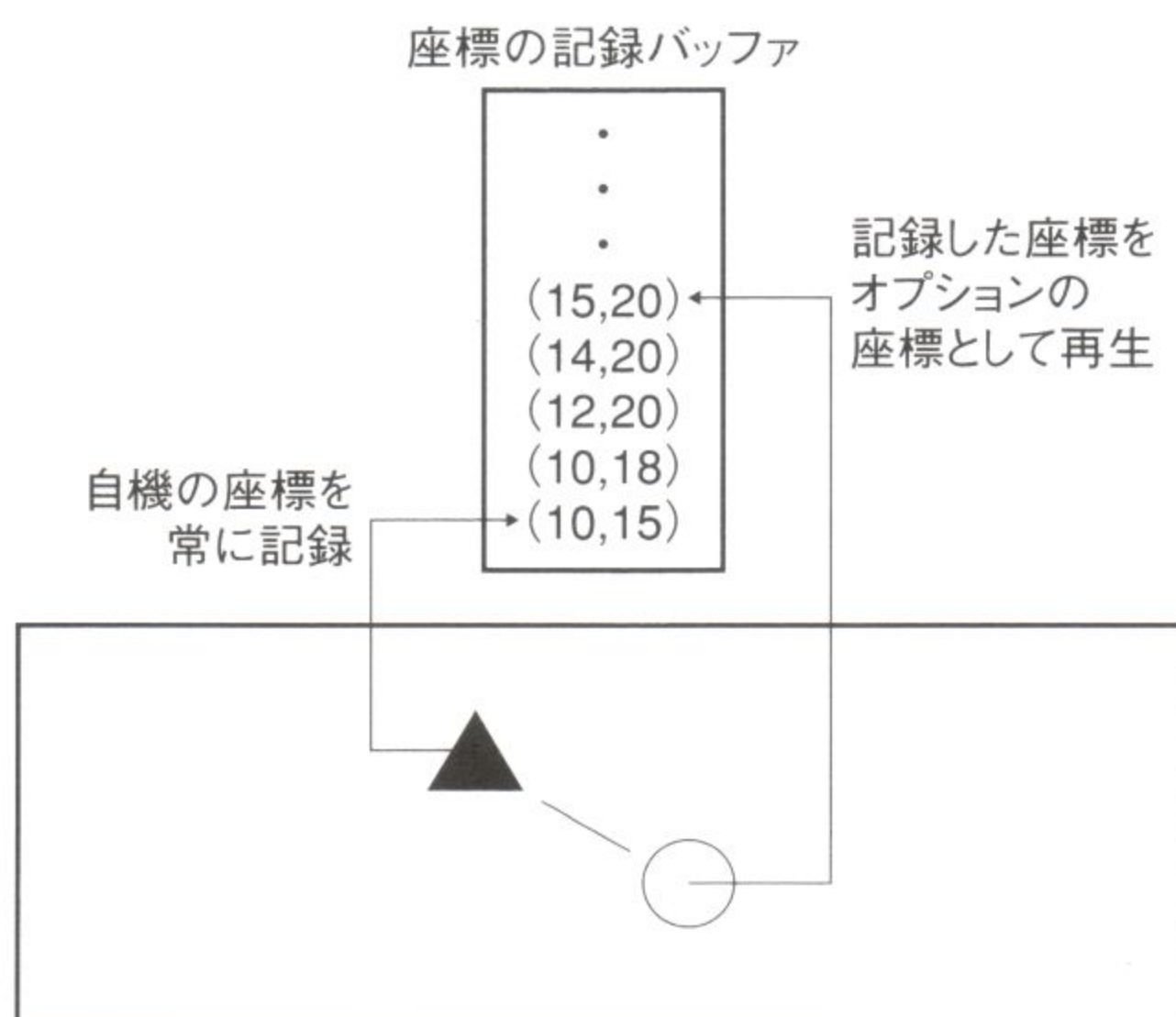
図6 －15－1 常に自機を追いかけるオプションのイメージ図

自機の動いた後を、数珠繋ぎで、追いかけるように動く

図6 －15－2 記録した座標をオプションが再生するイメージ図

座標の記録バッファ
・
・
・
(15,20)
(14,20)
(12,20)
(10,18)
(10,15)
記録した座標を
オプションの
座標として再生
自機の座標を
常に記録

自機を追いかけるオプションのプログラム

初期化

はじめに初期化処理で、自機タスクとオプションタスクを作成しますが、この部分は[6－14]の初期化部分とほぼ同じなので割愛します。

詳細は[6－14]を参照してください。

自機タスク

初期化処理の次は、自機タスクの処理です。

自機タスクは自機の移動処理と、自機の座標の記録を行ないます。

座標を記録する位置はタスクワーク上の変数PosPointerで管理しています。

自機の座標は過去64フレーム分を記録しており、64フレームを越えたら古い部分の履歴を上書きしていきます。このためPosPointerの値は0～63の値を常に維持します。

オプションタスク

次にオプションタスクの処理です。

基本的には、自機タスクの座標履歴を調べ、常にその座標を再生してやります。

ただこの際、IDに合わせて、引用する履歴の位置をずらしてやらなければなりません。

どの程度ずらすかで、オプションの移動する幅が変わります。

```
//オプションのIDから表示する座標の履歴を取得
option_pos_pointer = work->Myship->PosPointer + (work->OptionID * 16 + 15);
option_pos_pointer %= 64;
```

サンプルでは16フレームずらしていますが、ずらす際に範囲を超えて履歴をアクセスしないように注意が必要です。

LIST 6 －15－1

```
typedef struct{
    SPRITE          sprt;
    int             PosPointer;
    D3DXVECTOR2     PosBuff[64];        //座標履歴
} EX06_15_MY_SHIP;

typedef struct{
    SPRITE              sprt;
```

```
    EX06_15_MY_SHIP*  Myship;                  //自機情報
    int               OptionID;
} EX06_15_OPTION;


void exec06_15_option(TCB* thisTCB)
{
    EX06_15_OPTION* work = (EX06_15_OPTION*)thisTCB->Work;
    int option_pos_pointer;

    //オプションのIDから表示する座標の履歴を取得
    option_pos_pointer = work->Myship->PosPointer + (work->OptionID * 16 + 15);
    option_pos_pointer %= 64;

    work->sprt.X = work->Myship->PosBuff[ option_pos_pointer ].x + 16;
    work->sprt.Y = work->Myship->PosBuff[ option_pos_pointer ].y + 16;

    SpriteDraw(&work->sprt,1);
}

void exec06_15_my_ship(TCB* thisTCB)
{
#define MOVE_SPEED  8.0
    EX06_15_MY_SHIP* work = (EX06_15_MY_SHIP*)thisTCB->Work;

    //キー入力による移動
    if( g_InputBuff & KEY_UP    ) work->sprt.Y -= MOVE_SPEED;
    if( g_InputBuff & KEY_DOWN  ) work->sprt.Y += MOVE_SPEED;
    if( g_InputBuff & KEY_RIGHT ) work->sprt.X += MOVE_SPEED;
    if( g_InputBuff & KEY_LEFT  ) work->sprt.X -= MOVE_SPEED;

    work->PosPointer++;
    work->PosPointer %= 64;
    //移動座標を履歴に記録
    work->PosBuff[work->PosPointer].x = work->sprt.X;
    work->PosBuff[work->PosPointer].y = work->sprt.Y;

    SpriteDraw( &work->sprt, 0);
}
```

6-16 オプションの動き 3

自機を追いかける

3つ目のオプションは応用編として、自機を追いかける動きを作成してみましょう。

これは動き自体は、[6－15]とよく似ており、またオプションの動きとしても最もよく知られた物だと思われます。

つまり、常に自機を追いかけるのではなく、自機が動いた時だけ、自機を追従する動作をします。

オプションの動き 2 との違い

実際、基本となる処理も[6－15]とよく似ており、リアルタイムに動きを記録、追従していた処理を、自機の移動、すなわち、キー入力に合わせて行なう形になっています。

なお、オプションタスクに関しては処理はまったく同じです。詳細は[6－15]を参照してください。

LIST 6 －16－1

```
typedef struct{
    SPRITE          sprt;
    int             PosPointer;
    D3DXVECTOR2     PosBuff[64];                //座標履歴
} EX06_16_MY_SHIP;

void exec06_16_my_ship(TCB* thisTCB)
{
    EX06_16_MY_SHIP* work = (EX06_16_MY_SHIP*)thisTCB->Work;
    BOOL move_flag = FALSE;

    //キー入力による移動
    if( g_InputBuff & KEY_UP    )
    {
        work->sprt.Y -= MOVE_SPEED;
        move_flag = TRUE;
    }
```

```
    if( g_InputBuff & KEY_DOWN  )
    {
        work->sprt.Y += MOVE_SPEED;
        move_flag = TRUE;
    }
    if( g_InputBuff & KEY_RIGHT )
    {
        work->sprt.X += MOVE_SPEED;
        move_flag = TRUE;
    }
    if( g_InputBuff & KEY_LEFT  )
    {
        work->sprt.X -= MOVE_SPEED;
        move_flag = TRUE;
    }

    if( move_flag )
    {//移動が行われた時のみ履歴を更新
        work->PosPointer++;
        work->PosPointer %= 64;
        //移動座標を履歴に記録
        work->PosBuff[work->PosPointer].x = work->sprt.X;
        work->PosBuff[work->PosPointer].y = work->sprt.Y;
    }
    SpriteDraw( &work->sprt, 0);
}
```

こうした履歴を再生するタイプのオプションは、オプションタスクではなく、自機タスクの履歴の記録時に、再生ポイントや記録座標を操作する事によって、いろんな動きを行なう事が出来ます。

サンプルを改造して、色々試してみてください。

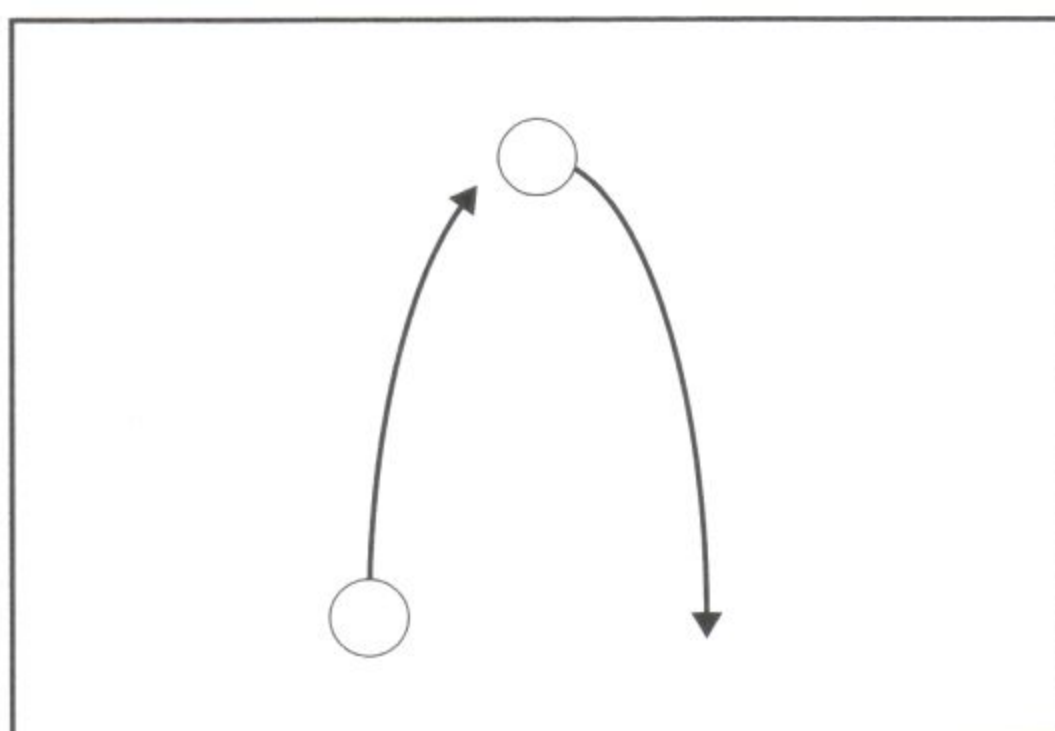
6-17 ジャンプをする

ジャンプのアルゴリズム

ジャンプはアクションゲームでは必ずといっていいほど使用されます。

ジャンプもゲームの種類や内容によって、色々なタイプのアルゴリズムが存在します。

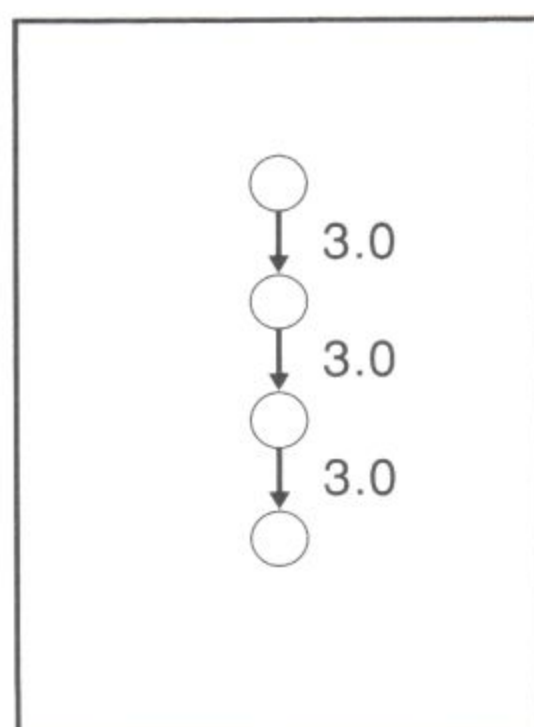
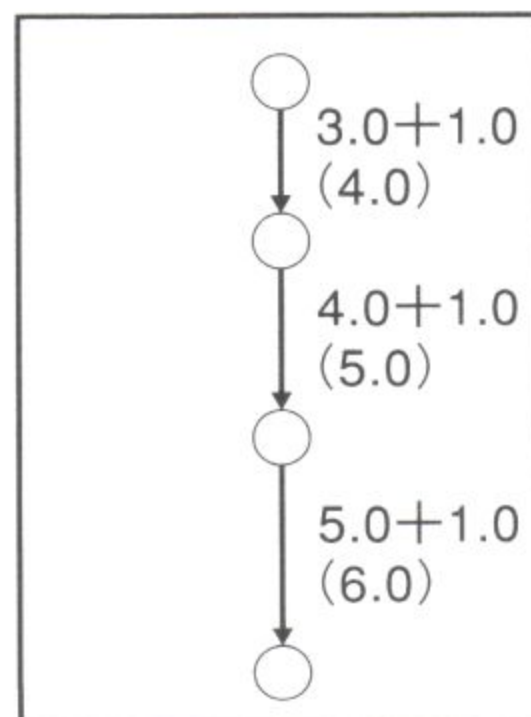
ここでは一番よく使用されていると思われる、放物線を使用したジャンプの動きを再現してみましょう。

図6 ―17―1 放物線のイメージ図

放物線の動きとは、落下、すなわち等加速度運動のことを言います。

この等加速度運動、完全に制御を行なおうとすると、微分だとか色々ややこしい話が出てくるのですが、放物線に沿った移動をするだけなら単純です。

文字通り、等速度で動いている物に、加速値を加えて「加速」してやります。

この際、加速する値で落下速度が変わるため、この加速値の事を重力値とも呼びます。

図6 ―17―2 等速運動物に加速を加えるイメージ図

ジャンプのプログラム

ジャンプの初期化

実際にリストを見ていきましょう。

初期化処理が終わったら、キー入力をチェックし入力があれば、ジャンプの初期化処理を行ないます。

初期化の内容はジャンプの初期上昇値の設定と、ジャンプ中である事をチェックするフラグのクリアです。

もしジャンプ中であれば、等加速度運動の処理を行ないます。

着地の判定

着地の判定は、現在の座標が地面の座標（サンプルではSCREEN_HEIGHT/2）を越えているかどうかでチェックします。

着地後はフラグのクリアと、地面への“めり込み”を防ぐための座標の再設定を行ない、ジャンプを終了します。

LIST 6 －17－1 ジャンプするには

```
typedef struct{
    SPRITE          sprt;
    float           Acc;
    int             JumpFlag;
} EX06_17_STRUCT;


void init06_17(TCB* thisTCB)
{
    EX06_17_STRUCT* work = (EX06_17_STRUCT*)thisTCB->Work;
    //使用するテクスチャの読み込み
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥human.png",&g_pTex[0] );

    //座標初期化
    work->sprt.X = SCREEN_WIDTH  / 2;
    work->sprt.Y = SCREEN_HEIGHT / 2;

}
```

```
void exec06_17(TCB* thisTCB)
{
#define JUMP_POWER   10.0   //ジャンプ力
#define GRAVITY   0.25      //重力値

    EX06_17_STRUCT* work = (EX06_17_STRUCT*)thisTCB->Work;

    //ジャンプ中は反応しない
    if( !work->JumpFlag )
    {//着地時の処理
        //キー入力でジャンプ処理
        if( g_DownInputBuff & KEY_Z     )
        {//ジャンプ開始時処理
            work->Acc = -JUMP_POWER;
            work->JumpFlag = true;
        }
     }

    if( work->JumpFlag )
    {
        work->Acc += GRAVITY;
        work->sprt.Y += work->Acc;

        //着地したらジャンプ処理終了
        if( work->sprt.Y > SCREEN_HEIGHT / 2)
        {//座標が、地面座標より下ならば、着地したとみなす
            work->sprt.Y =  SCREEN_HEIGHT / 2;
            work->JumpFlag = false;
        }
    }

    SpriteDraw( &work->sprt, 0);
}
```

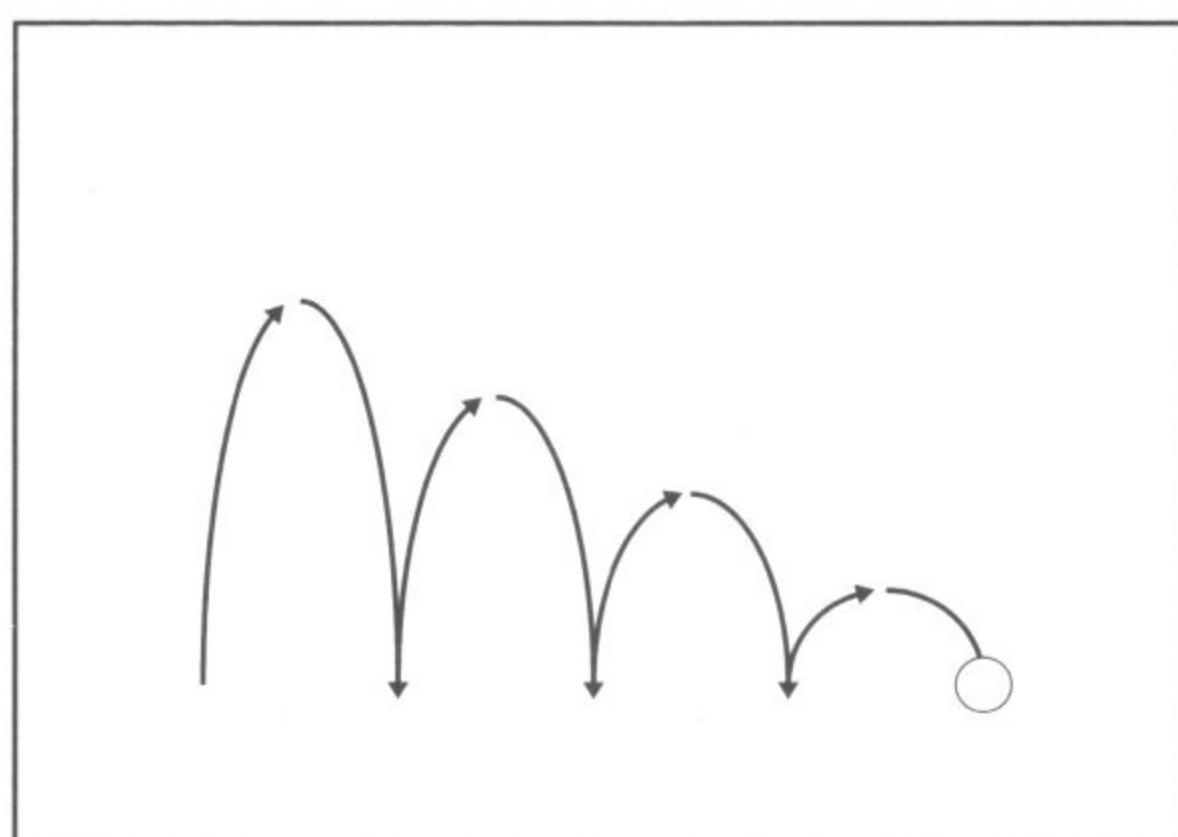
6-18 地面に対してバウンドをする

バウンドの演出

地面に対してバウンドをする処理を行なってみましょう。

バウンドは攻撃の方法や、演出、敵の動きなど、様々な場所で使われます。

動きのイメージは、地面で跳ねるボールを想像してもらえればわかりやすいでしょうか。

図6 ー18ー1 バウンドのイメージ図

地面にぶつかる度に、減衰していく

バウンドのプログラム

基本的には、ジャンプの処理とよく似ています。

ジャンプと異なる点は、地面に接触した時の処理です。

```
work->sprt.Y -= work->Acc;
work->Acc = -( work->Acc - BOUND_DECR );      //バウンド時に減衰する
```

ジャンプでは、地面に接触した時に終了しましたが、バウンドでは地面に対して反発します。

反発の処理は、加速値の符号を反転させる事で実現します。またその際、バウンド時の減衰処理も行ないます。

反発した物体は、上昇後、また落下し、再度地面に接触します。その際も同様に減衰処理を行ないます。

このように何度かバウンドを繰り返すと、最終的に、ほとんどバウンドをしなくなります。そうなったら、処理を元に戻しバウンド処理を終了します。

LIST 6 －18－1

```
typedef struct{
    SPRITE          sprt;
    float           Acc;
    int             BoundFlag;
} EX06_18_STRUCT;

void init06_18(TCB* thisTCB)
{
    EX06_18_STRUCT* work = (EX06_18_STRUCT*)thisTCB->Work;
    //使用するテクスチャの読み込み
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥human.png",&g_pTex[0] );

    //座標初期化
    work->sprt.X = SCREEN_WIDTH  / 2;
    work->sprt.Y = SCREEN_HEIGHT / 2;

}

void exec06_18(TCB* thisTCB)
{
#define JUMP_POWER  10.0      //ジャンプ力
#define GRAVITY     0.25      //重力値
#define BOUND_DECR  1.0       //バウンド時の減衰

    EX06_18_STRUCT* work = (EX06_18_STRUCT*)thisTCB->Work;

    //バウンド中は反応しない
    if( !work->BoundFlag )
    {
        //キー入力でジャンプ処理
        if( g_DownInputBuff & KEY_Z    )
        {//バウンド開始時処理
            work->Acc = -JUMP_POWER;
            work->BoundFlag = true;
```

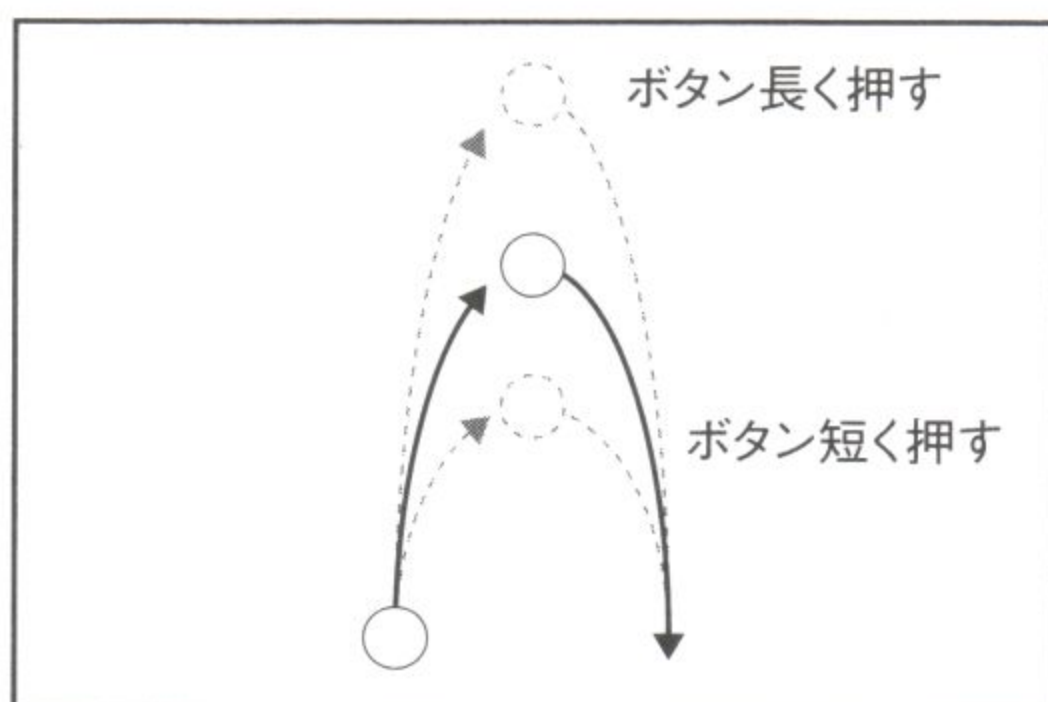
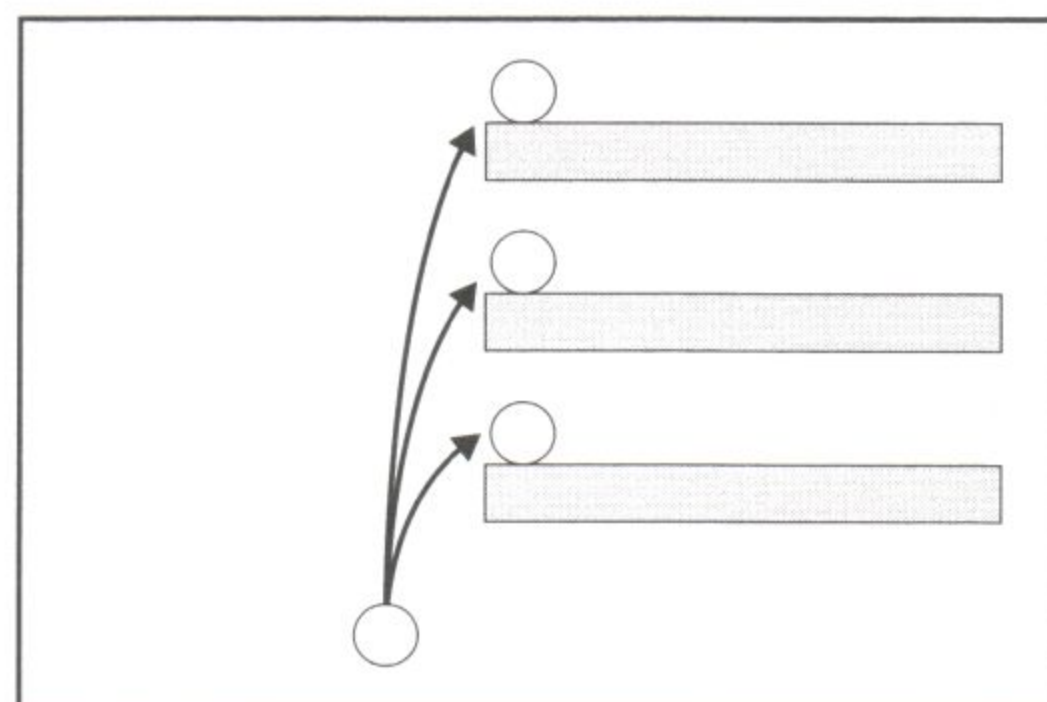
```
        }
    }

    if( work->BoundFlag )
    {
        work->Acc += GRAVITY;
        work->sprt.Y += work->Acc;

        //地面に再接触したら、バウンドする
        if( work->sprt.Y > SCREEN_HEIGHT / 2 )
        {
            work->sprt.Y -= work->Acc;
            //バウンド時に減衰する
            work->Acc = -( work->Acc - BOUND_DECR );

            if( work->Acc > -1.0 )
            { //殆どバウンドしなくなったら終了
                work->sprt.Y =  SCREEN_HEIGHT / 2;
                work->BoundFlag = false;
            }
        }
    }

    SpriteDraw( &work->sprt, 0);
}
```

6-19 ボタンを押す長さでジャンプの高さを変える

ジャンプの高さを調整できるようにする

アクションゲームではボタンを押す長さで、ジャンプの高さを変えるケースがよくあります。

細かい高さの調整が可能になる事で操作性が向上すると共にゲーム性に幅が出て、面白さが増すためです。

図6 ー19ー1 ボタンでジャンプの高さを変更し操作性の上がるするイメージ図

ボタンを押す長さによってジャンプの高さの操作が可能になると

好きな場所にいけるなど、操作の幅が広がり、ゲーム性が増す

実際のプログラム

実際に高さを変える処理ですが、ボタンを押している時と、離している時の、重力値を変更する事で行ないます。

リストを見ていきましょう。

初期化とジャンプの処理

初期化処理は、通常のジャンプの時の処理と同一です。

ジャンプ時の初期化は、移動時に加える重力値が途中で変えられるように変更しています。

さて、肝心のボタンの処理ですが、ボタンが離された時に、加える重力値を変更してやります。

注意しなければならない点

注意する点として、落下時には重力値を変更しない事、また、変更する時は速度に応じて重力値を変えてやる事ことが挙げられます。

この2点が守られないと、非常に違和感のあるジャンプになってしまいます。

なお、もしボタンが押しっぱなしであれば、通常のジャンプと同じ処理になります。

後は着地の処理ですが、この点は通常のジャンプとまったく同じ処理になります。

LIST 6 —19—1 ボタンを押す長さでジャンプの高さを変える

```
typedef struct{
    SPRITE          sprt;
    float           Acc;
    int             JumpFlag;
    float           Gravity;
} EX06_19_STRUCT;
void init06_19(TCB* thisTCB)
{
    EX06_19_STRUCT* work = (EX06_19_STRUCT*)thisTCB->Work;
    //使用するテクスチャの読み込み
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥human.png",&g_pTex[0] );

    //座標初期化
    work->sprt.X = SCREEN_WIDTH  / 2;
    work->sprt.Y = SCREEN_HEIGHT / 2;

}

void exec06_19(TCB* thisTCB)
{
#define JUMP_POWER  10.0        //ジャンプ力
#define PUSH_GRAVITY  0.25      //押下時の重力値
#define FALL_RATE     8.0       //落下速度の計算値

    EX06_19_STRUCT* work = (EX06_19_STRUCT*)thisTCB->Work;


    //ジャンプ中は反応しない
    if( !work->JumpFlag )
    {//着地時の処理
        //キー入力でジャンプ処理
        if( g_DownInputBuff & KEY_Z     )
        {//ジャンプ開始時処理
```

```
            work->Acc = -JUMP_POWER;
            work->JumpFlag = true;
            work->Gravity = PUSH_GRAVITY;
        }
     }

    if( work->JumpFlag )
    {
        //ジャンプボタンが離されたか?
        if( g_UpInputBuff & KEY_Z  )
        {
            //上昇中だけ処理
            if( work->Acc < 0 )
            {//ボタンが離された時の上昇速度に応じて落下速度を変える
                work->Gravity = -work->Acc / FALL_RATE;
            }
        }

        work->Acc += work->Gravity;
        work->sprt.Y += work->Acc;
        //着地したらジャンプ処理終了
        if( work->sprt.Y > SCREEN_HEIGHT / 2)
        {
            work->sprt.Y =  SCREEN_HEIGHT / 2;
            work->JumpFlag = false;
        }
    }

    SpriteDraw( &work->sprt, 0);
}
```

6-20 好きな高さと時間でジャンプする

制作途中でジャンプの高さを調整したいとき

アクションゲームではジャンプの細かい調整はよく行なわれます。

重力値や時間を少しづつ変更してもいいのですが、場合によっては非常に大変な作業になります。

その際、座標や時間を直接指定できればとても便利です。

実際のプログラム

実際のリストですが、メインのジャンプ処理自体は通常のジャンプ処理とまったく同じです。

ただ、ジャンプの開始時に、初期上昇値と重力値を取得するため、関数GetJumpPowerを呼び出しています。

関数の中身ですが、時間と高さから直接、加速値(重力値)を導く式になっています。

初期上昇値は、取得した値に時間を掛ける事で得られます。

また加速値は、関数から直接得られるので、そのまま設定するだけでOKです。

直感的には少しわかりにくいかも知れませんが、設定するパラメータを色々調整して試してみてください。

LIST 6 －20－1 好きな高さでジャンプする

```
typedef struct{
    SPRITE          sprt;
    float           Acc;
    float           Gravity;
    int             JumpFlag;
} EX06_20_STRUCT;

void init06_20(TCB* thisTCB)
{
    EX06_20_STRUCT* work = (EX06_20_STRUCT*)thisTCB->Work;
    //使用するテクスチャの読み込み
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥human.png",&g_pTex[0] );

    //座標初期化
```

```
    work->sprt.X = SCREEN_WIDTH  / 2;
    work->sprt.Y = SCREEN_HEIGHT / 2;
}


float GetJumpPower( float Time, float Height )
{
    return Height / (Time * (Time + 1.0) * (1.0 / 2.0) );
                                        //高さ ÷ 時間×時間+1 × 1/2
}


void exec06_20(TCB* thisTCB)
{
#define JUMP_POS_A  256.0   //ジャンプ高さ
#define JUMP_POS_B  128.0   //ジャンプ高さ
#define JUMP_TIME_A 30.0    //頂点までの時間
#define JUMP_TIME_B 60.0    //頂点までの時間

    EX06_20_STRUCT* work = (EX06_20_STRUCT*)thisTCB->Work;

    //ジャンプ中は反応しない
    if( !work->JumpFlag )
    {
        //キー入力でジャンプ処理
        if( g_DownInputBuff & KEY_Z     )
        {//ジャンプ開始時処理A
            work->Acc = -(GetJumpPower(JUMP_TIME_A , JUMP_POS_A) * 
JUMP_TIME_A);
            work->Gravity = GetJumpPower(JUMP_TIME_A , JUMP_POS_A);
            work->JumpFlag = true;

        }
        if( g_DownInputBuff & KEY_X     )
        {//ジャンプ開始時処理B
            work->Acc = -(GetJumpPower(JUMP_TIME_B , JUMP_POS_B) * 
JUMP_TIME_B);
            work->Gravity = GetJumpPower(JUMP_TIME_B , JUMP_POS_B);
            work->JumpFlag = true;
        }
    }
```

```
    if( work->JumpFlag )
    {
        work->Acc += work->Gravity;
        work->sprt.Y += work->Acc;

        //着地したらジャンプ処理終了
        if( work->sprt.Y > SCREEN_HEIGHT / 2)
        {
            work->sprt.Y =  SCREEN_HEIGHT / 2;
            work->JumpFlag = false;
        }
    }

    SpriteDraw( &work->sprt, 0);
}
```

6-21 スクロールすると敵が出てくるようにする

スクロールポイント

スクロールによる敵の出現について考えてみましょう。

一般にスクロールをするゲームでは、スクロールしてゲームが進行すると、それに合わせて敵が出現します。

この際に出てくる敵は、スクロールに合わせてパターン化されていたり、最後までスクロールすると、ボスキャラが登場したりします。

このような処理を行なうためには、どの程度進行したかを示す位置「スクロールポイント」に合わせて、出現の処理を行ないます。

この際、わざわざスクロールポイントを用意せず、スクロールの座標を用いても良いのですが、応用がしにくいため、専用の変数を設けるのが一般的です。

スクロールポイントのプログラム

では、実際のプログラムです。

サンプルでは、自動的にスクロールする背景が進行するにしたがって、上からボールが降ってくるようになっています。

スクロール停止後、Z キーで最初に戻ります。

配列データの説明

さて、実際の処理解説の前に、まずはデータの説明をします。

配列データのscroll_table を見てください。これは、スクロールポイントがこの値に達した時に処理を行なうためのデータの配列です。

例えば最初の値に30 とありますが、これは最初のボールはスクロールポイントが30 になった時にボールが発生する、という事です。

メイン処理

次はようやくメインの処理解説です。まず、いきなりですが、スクロールポイントをチェックしています。

データの解説であったように、ここでスクロールポイントと、発生データの比較を行なっています。

比較後、もしボールが発生したならば、次発生値との比較を行なうため、発生データのIndex 値を進めています。

スクロール座標と、スクロールポイント

スクロールによる出現に関しては以上なのですが、最後に背景スクロールに関して触れておきます。

スクロール座標と、スクロールポイントは本来別の値なのですが、ここでは同期の例として、スクロールポイントをスクロール座標に変換しています。

このようにしておくと、何らかの処理やボスキャラの出現時などで、スクロール座標と敵の出現座標が合わないといった事がなくなります。

LIST 6－21－1 スクロールすると敵が出てくるようにする

```
typedef struct{
    BACK_GROUND     BG;
    int             ScrollPoint;
    int             DataIndex;
} EX06_21_STRUCT;

typedef struct{
  SPRITE      Sprt;
  int         InitFlag;
} EX06_21_BALL;

void exec06_21_ball(TCB* thisTCB)
{
#define MOVE_SPEED 16
    EX06_21_BALL* work = (EX06_21_BALL*)thisTCB->Work;

    if( work->InitFlag != TRUE )
    {//初期化
        //座標の設定
        work->Sprt.X = rand() % SCREEN_WIDTH;
        //初期化終了のフラグ
        work->InitFlag = TRUE;
    }
    //移動処理、上から落下する
    work->Sprt.Y += MOVE_SPEED;

    SpriteDraw( &work->Sprt ,0);
}
```

```
void init06_21(TCB* thisTCB)
{
    //使用するテクスチャの読み込み
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥0055.png",&g_pTex[0] );
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥BG.png",&g_pTex[1] );
}

void exec06_21(TCB* thisTCB)
{
#define MOVE_SPEED 8
    EX06_21_STRUCT* work = (EX06_21_STRUCT*)thisTCB->Work;
    char str[256];
int scroll_table[] =
{//出現データ
    30,  50,  60,  70,
    75,  80, 100, 110,
   111, 112, 113, 114,
   120, 150, 175, 180,
   188, 192, 224, 276,
   280, 292, 300, 304,
   309, 312, 322, 340,
   350, 389, 392, 448,
   -1,
};


    //出現データとスクロールポイントを比較
    if( work->ScrollPoint == scroll_table[ work->DataIndex ] )
    {//一致したら出現
        TaskMake( exec06_21_ball, 0x2000 );

        //次の比較データ
        work->DataIndex++;
    }

    //スクロール処理
```

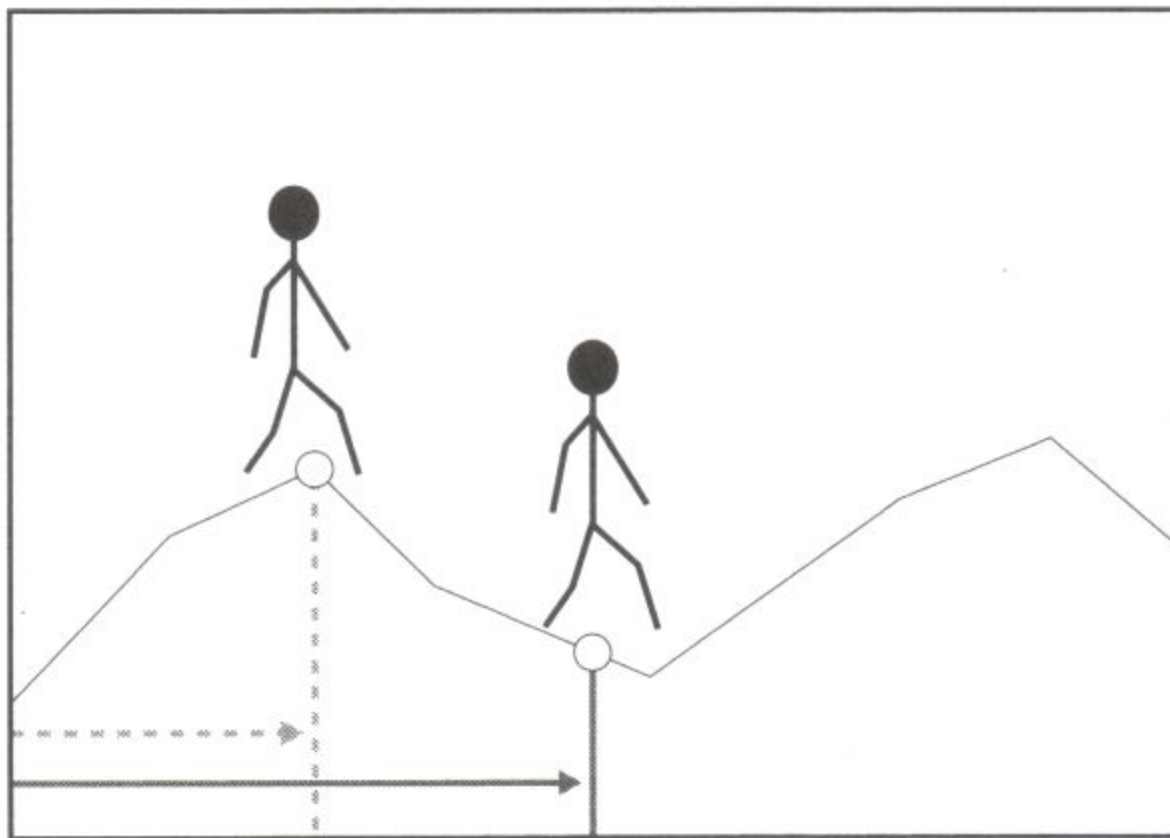
```
        //もしキー入力があれば初期化する
        if( g_DownInputBuff & KEY_Z     )
        {
            work->ScrollPoint = 0;
            work->DataIndex = 0;
        }

        //スクロールポイントを座標に変換
        work->BG.X = -work->ScrollPoint;
        //画面端に到達していなければスクロールさせる
        if(work->BG.X >  -384 ) work->ScrollPoint++;

        BGDraw(&work->BG,1);

        sprintf( str,"'Z'KEY Scroll Reset!!¥n SCROLL POINT %d",work->ScrollPoint );
        FontPrint( 128,128, str);

    }
```

6-22 高低差のある地形を移動するには

起伏のある地面

高低差のある地面を表現してみましょう。

一般的な地面と違い、高低差、すなわち起伏のある地面は、プレイヤーの横座標に応じて地面の高さが変わります。

図6 －22－1 横座標に応じて高さが変わる図

こういった地面は起伏の高さのデータ、要するに地形データを用意してやる事で、容易に実現する事が出来ます。

地形データによる地面

早速プログラムを見ていきましょう。サンプルは、方向キーで左右にキャラを動かす事で、同時に地形に合わせても上下移動も行なうものです。

はじめに初期化として地形データを作成します。地形データは配列で定義され、画面の横幅分だけ用意されます。

ただ、画面幅（640ドット）分ものデータを持つのは容量的にも大きく、また、通常はそこまでの精度も要求される事はないため、サンプルでは4ドット単位でデータを作成しています。

サンプルのデータはsin関数で作成され、その値は基準の地面の位置（定数HEIGHT_BASE_POS)に上下の幅（UP_DOWN）を加える事で得ています。

地形データ上の移動

地形データができれば、後はその上を移動するだけです。

地形データ配列から高さデータを得るには、プレイヤーの左右の位置、すなわちX座標を、地形データ配列の添え字として使用するだけです。

ただし、座標は浮動小数点なので、int型へとキャストを行なっています。

高さデータを得たら、プレイヤーY座標をその座標に合わせてやれば、上下の移動が実現できます。

地形データ上の表示

なお、その他の処理として、一番最後で、地形の表示を行なっています。これは地形データの精度(4ドット)毎に4ドット幅のスプライトを表示する事で行なっています。

単純に地形データ配列の位置にスプライトを表示するだけなので、難しい所は無いでしょう。

LIST 6－22－1 高低差のある地形を移動するには

```
#define DATA_SIZE 4
#define HEIGHT_DATA (SCREEN_WIDTH/DATA_SIZE)
#define HEIGHT_BASE_POS (SCREEN_HEIGHT-64)
#define UP_DOWN 32
#define MOVE_SPEED 4.0

#define HUMAN_CENTER_X 32
#define HUMAN_CENTER_Y 128

typedef struct{
    SPRITE   Human;
    SPRITE   GrandBar;
    float    X;
    float    Y;
    float    HeightData[ HEIGHT_DATA ];
} EX06_22_STRUCT;

void init06_22(TCB* thisTCB)
{
    EX06_22_STRUCT* work = (EX06_22_STRUCT*)thisTCB->Work;
    int loop;
    float sin_wave = 0;
```

```
    //使用するテクスチャの読み込み
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥human.png",&g_pTex[0] );
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥bar.png",&g_pTex[1] );

    //座標の設定
    work->X = SCREEN_WIDTH / 2;

    for( loop=0;loop < HEIGHT_DATA; loop++)
    { //sin関数を利用して上下のある地形を生成
        work->HeightData[loop] =
            sin( sin_wave ) * UP_DOWN + HEIGHT_BASE_POS;
        sin_wave += 0.125;
    }

}

void exec06_22(TCB* thisTCB)
{
    EX06_22_STRUCT* work = (EX06_22_STRUCT*)thisTCB->Work;
    int loop;

    //キー入力による移動
    if( g_InputBuff & KEY_RIGHT ) work->X += MOVE_SPEED;
    if( g_InputBuff & KEY_LEFT  ) work->X -= MOVE_SPEED;
    //移動範囲を画面内にする
    if(work->X >= SCREEN_WIDTH - DATA_SIZE)
        work->X = SCREEN_WIDTH - DATA_SIZE;
    if(work->X <  0)work->X = 0;

    //X座標から、地面の高さデータを取得
    work->Y = work->HeightData[(int)work->X/DATA_SIZE];

    //表示座標を設定
    work->Human.X = work->X - HUMAN_CENTER_X;
    work->Human.Y = work->Y - HUMAN_CENTER_Y;

    //キャラクターの表示
```

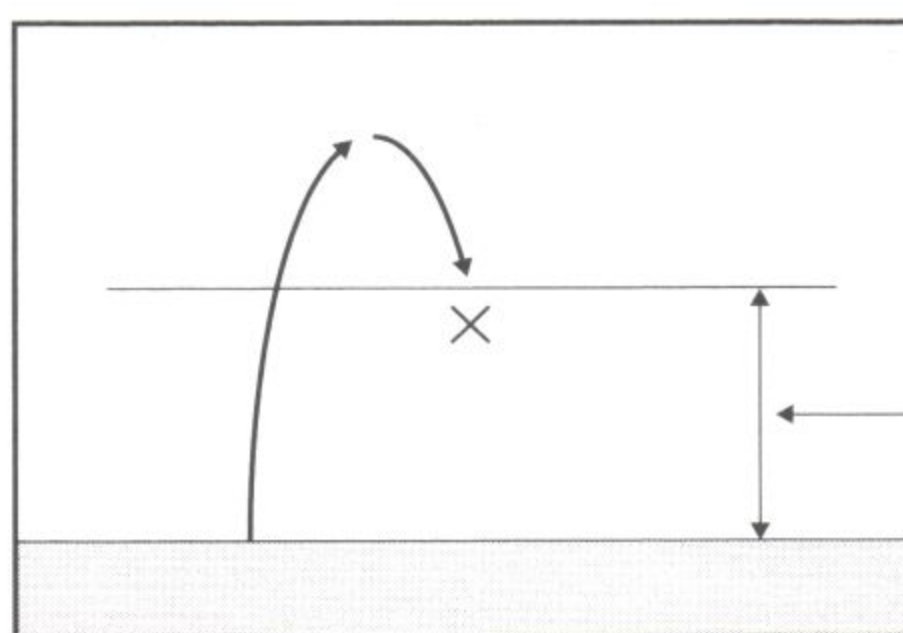
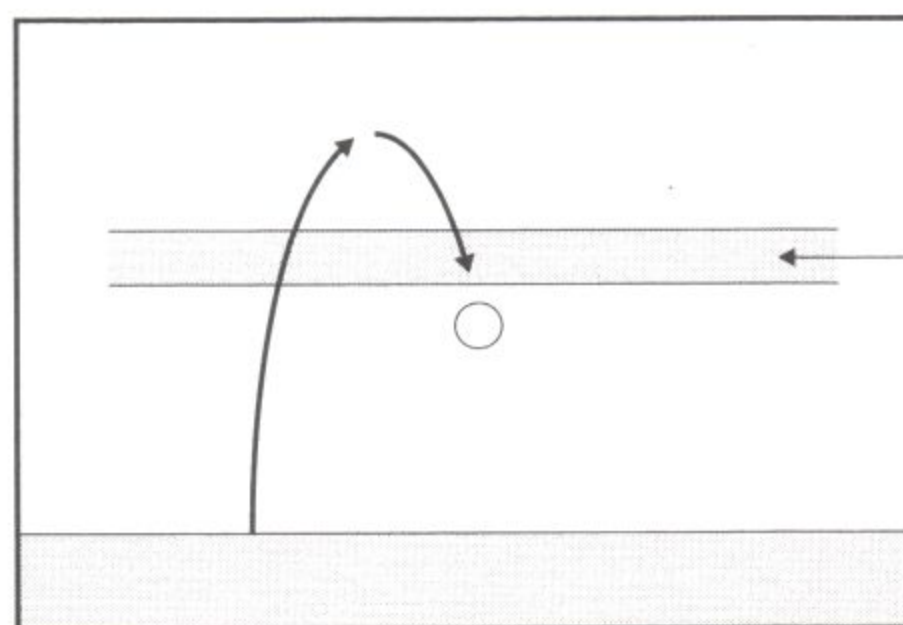
```
    SpriteDraw( &work->Human,0);

    for( loop=0;loop < HEIGHT_DATA; loop++)
    { //地形データの表示
        work->GrandBar.X = loop * DATA_SIZE;
        work->GrandBar.Y = work->HeightData[loop];
        SpriteDraw( &work->GrandBar,1);
    }
}
```

6-23 段差のある地面に着地するには

違う高さの場所へのジャンプ

アクションゲームでは、ジャンプで移動が可能な複数の地面があります。
この段差のある地面に着地するには、どうしたら良いかを考えてみましょう。

基本的なジャンプに対する着地は、[6－17]で解説したように、Y座標が地面の座標値以上であれば、判定する事が出来ます。
ところが、段差のある地面は、ジャンプする座標よりも上方に位置する事が多く、このままでは使用できません。
また、段差に着地したのであれば、当然そこから「降りる」処理も必要になるため、それに対処する事も難しそうです。
結論から先に書いてしまいますと、段差の地面の座標だけではなく「段差の領域」を定義し、そこに「落下中のキャラ」の座標が差し掛かった時に着地の処理を行ないます。
降りる時は、着地の処理を行なわずにそのまま、落下の処理を行なえば問題なく降りる事が出来ます。

図6－23－1 段差のある地面のイメージ図と、領域を定義してチェックするイメージ図

階層の座標だけだと、何処まで着地の判定を行なうのか分からない

そこで、判定が可能なように着地が可能な領域を設ける

違う高さの場所へジャンプするときのプログラム

では、実際のプログラムを見ていきましょう。

サンプルはＺキーでジャンプして上の段差に着地し、方向キーの下で段差から降りる処理を行ないます。

メイン処理

初期化後、メイン処理では、まずジャンプ状態のチェックを行ないます。

非ジャンプ状態（着地中の時）は主にキー入力を行ないます。

Ｚキーでジャンプの開始処理、下キーで落下の開始処理を行ないます。

着地処理

次は、ジャンプ中の着地処理です。まず落下中かどうかを確認します。

もし下降中に段差の領域に接触していたならば、着地したとみなす事が出来ます。

段差領域自体ははマクロで定義されており、領域の縦幅の大きさは16です。

ただしチェックの際に、下キーを入力しているならば、段差から降りる処理であるため、着地の処理は行ないません。

着地処理自体は、キャラクターに、段差の座標と、ジャンプ中であるフラグをクリアするだけです。

地面への着地処理等は、[6－17]で解説しているので、そちらを参照してください。

なお複数の段差をもつ場合は、チェックする領域をデータ化し、ループなどで複数チェックするようにすると良いでしょう。

LIST 6 －23－1 段差のある地面に着地するには

```
typedef struct{
    SPRITE        sprt;
    SPRITE        Land;
    float         Acc;
    int           JumpFlag;
} EX06_23_STRUCT;

#define GRAND             (SCREEN_HEIGHT / 2)  //地面座標
#define LAND_AREA_TOP     GRAND - 144          //着地する段差領域
#define LAND_AREA_BOTTOM GRAND - 128
#define JUMP_POWER  10.0  //ジャンプ力
#define GRAVITY  0.25     //重力値

void init06_23(TCB* thisTCB)
```

```
{
    EX06_23_STRUCT* work = (EX06_23_STRUCT*)thisTCB->Work;
    //使用するテクスチャの読み込み
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥human.png",&g_pTex[0] );
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥block1.png",&g_pTex[1] );

    //座標初期化
    work->sprt.X = SCREEN_WIDTH  / 2;
    work->sprt.Y = GRAND;
    //着地する段差
    work->Land.X = SCREEN_WIDTH  / 2;
    work->Land.Y = 216;
}

void exec06_23(TCB* thisTCB)
{
    EX06_23_STRUCT* work = (EX06_23_STRUCT*)thisTCB->Work;
    int loop;
    RECT font_pos = { 128, 128, 640, 480, };

    //ジャンプ中は反応しない
    if( !work->JumpFlag )
    {//着地時の処理
        //キー入力でジャンプ処理
        if( g_DownInputBuff & KEY_Z     )
        {//ジャンプ開始時処理
            work->Acc = -JUMP_POWER;
            work->JumpFlag = true;
        }
        //もし上段差で、下キーを入力していたら、落下して段差から降りる
        if( g_DownInputBuff & KEY_DOWN   )
        {//落下処理開始
            work->Acc = 0;
            work->JumpFlag = true;
        }
     }

    if( work->JumpFlag )
```

```
    {
        work->Acc += GRAVITY;
        work->sprt.Y += work->Acc;

        //段差への着地処理、まず下降中かをチェックする
        if( work->Acc > 0.0 )
        {//もし下降中に段差のエリアに達したら、着地したとみなす
            if( work->sprt.Y > LAND_AREA_TOP && work->sprt.Y <
LAND_AREA_BOTTOM )
            {//ただし、下キーを入力しているならば、段差から降りる処理の為、何もしない
                if( !(g_InputBuff & KEY_DOWN))
                {//段差上への着地処理
                    work->sprt.Y =  LAND_AREA_TOP;
                    work->JumpFlag = false;
                }
            }
        }

        //地面の着地処理
        if( work->sprt.Y > GRAND)
        {//座標が、地面座標より下ならば、着地したとみなす
            work->sprt.Y =  GRAND;
            work->JumpFlag = false;
        }
    }

    SpriteDraw( &work->sprt, 0);

    //段差表示
    SpriteDraw( &work->Land, 1);

    g_pFont->DrawText( NULL,
        "Zキーでジャンプ、下キーで段差から降りる",
        -1, &font_pos,
        DT_LEFT, 0xffffffff);
}
```

6-24 移動する物体に着地するには

移動する物体での着地処理

移動する物体に着地する方法について考えてみましょう。

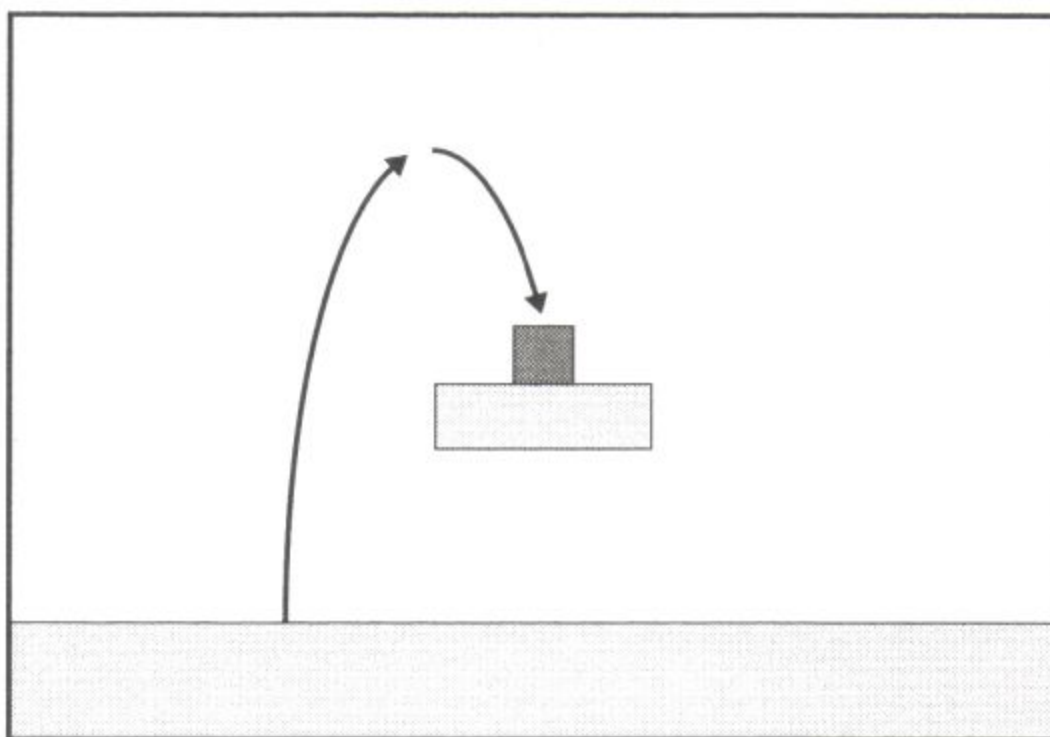
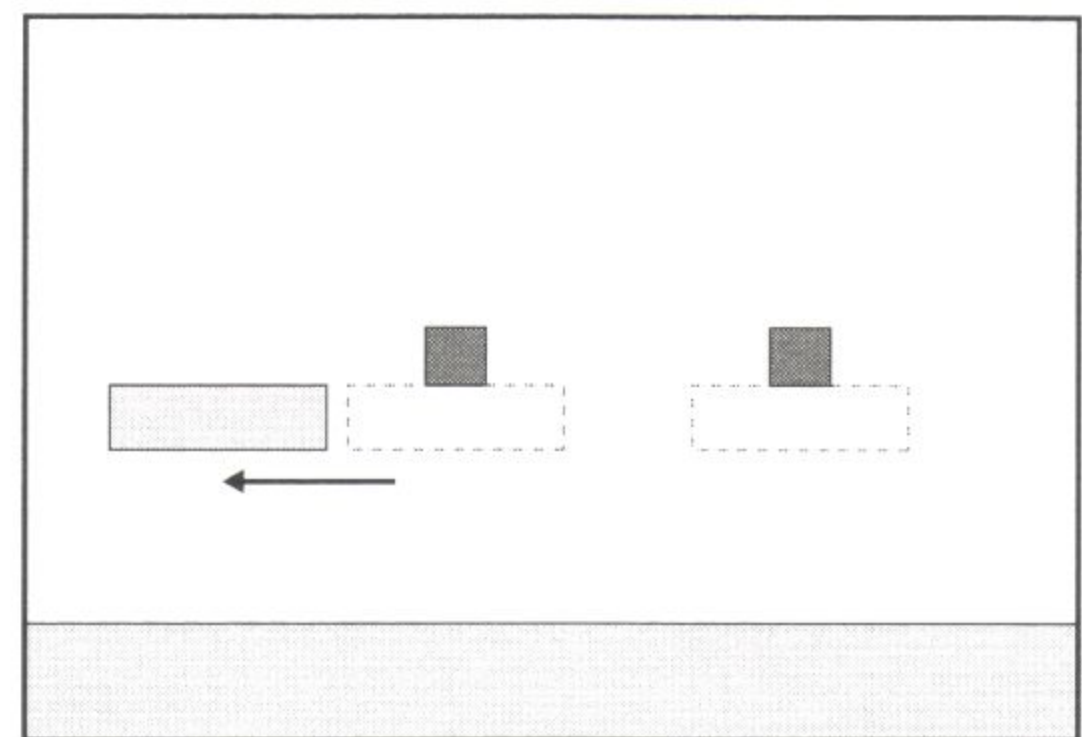
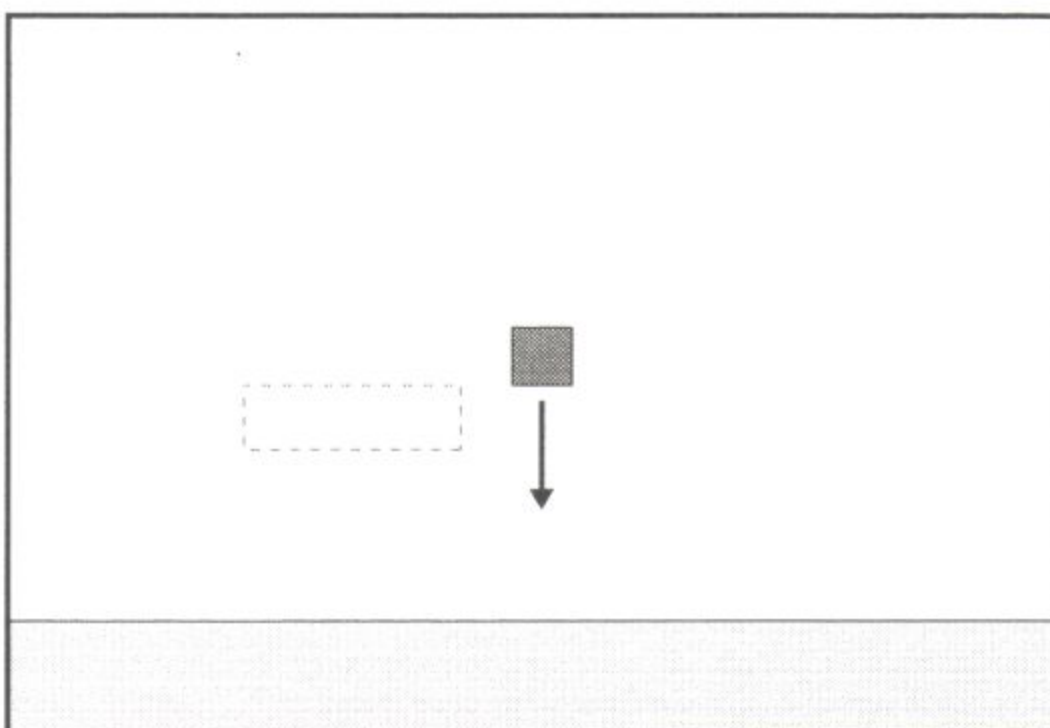
こういった動く物体に着地する処理は、一般的なアクションゲームでは必ずと言っていいほど出てくるシチュエーションで、必須の処理と言っても良いでしょう。

さて、処理方法ですが、基本的には段差の地面への着地処理の延長線上にあります。

すなわち、段差の領域の代わりに、着地する物体の大きさを着地判定に利用します。

しかし、そのままチェックするだけではダメです。

実際にはジャンプするキャラクターだけではなく、着地した物体も動いていますので、足元に物体があるかをチェックする処理も必要となります。

図6 ―24―1 移動する物体に着地後、足元の物体がなくなるケースを紹介

一旦物体に着地を行なっても

足元の物体が、急になくなったり強制的に移動をしてしまうなど、空中に浮いてしまう場合がある

その為、常に足元を見て落下可能かをチェックする処理が必要

実際のプログラム

それでは、実際の処理を見ていきましょう。

サンプルはZキーでジャンプ、方向キーでキャラを左右に動かします。

中央には、物体が左右に動いておりキャラを着地させる事が出来ます。

初期化とメイン処理

まず初期化処理ですが、ここら辺は通常のジャンプ処理とほぼ同じです。

次にメイン処理ですが、はじめにキャラの左右移動と、物体の左右移動を行ないます。

そして得られた位置から、物体の位置と大きさをRECT構造体に、キャラの足元の座標を変数に格納します。

着地時に足場があるかどうかチェック

次に着地時の処理ですが、Zキーを押してジャンプ開始をする時の処理は通常のジャンプと変わりません。

ただ、常に足元に物体があるとは限らない（常に着地状態とは限らない）ため、ここで先ほど取得した物体の位置と大きさを用いて足元のチェックを行ないます。

もし、足元に何も物体が無い場合は、落下処理を開始します。そうでない時は着地したままです。

着地処理

次は、着地の処理です。これも段差への着地と同様に落下中に、物体に接触判定を行なう事で着地したかどうかを判定する事が出来ます。

ここでも、先ほど取得した物体の位置と大きさを用いてチェックを行ないます。

もし接触していれば着地したとみなし、物体の座標から、キャラの着地座標を計算して設定します。

表示

最後に通常の地面の着地処理と表示処理を行ないます。

なお、このサンプルでは、複数の移動する物体には対応していません。

ですが、接触判定の部分はRECT構造体なので、この部分をデータ化するようにすれば、複数物体の処理もさほど難しくは無いと思います。

LIST 6 —24—1 移動する物体に着地するには

```
typedef struct{
    SPRITE          sprt;
    SPRITE          Land;
    float           LandMoveCount;
    float           Acc;
    int             JumpFlag;
} EX06_24_STRUCT;

#define GRAND               (SCREEN_HEIGHT / 2)  //地面座標
#define HUMAN_WIDTH         64
#define HUMAN_HEIGHT        128
#define LAND_MOVE_SPEED     0.05;
#define LAND_WIDTH          64
#define LAND_HEIGHT         16
#define JUMP_POWER  10.0    //ジャンプ力
#define GRAVITY  0.25       //重力値

void init06_24(TCB* thisTCB)
{
    EX06_24_STRUCT* work = (EX06_24_STRUCT*)thisTCB->Work;
    //使用するテクスチャの読み込み
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥human.png",&g_pTex[0] );
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥block1.png",&g_pTex[1] );

    //座標初期化
    work->sprt.X = SCREEN_WIDTH  / 2;
    work->sprt.Y = GRAND;
    //着地する物体
    work->Land.Y = 216;
}

void exec06_24(TCB* thisTCB)
{
    EX06_24_STRUCT* work = (EX06_24_STRUCT*)thisTCB->Work;
    int loop;
    RECT  land_rect;
    float my_pos_x;
```

```
    float my_pos_y;
    RECT font_pos = { 128, 128, 640, 480, };


    if( g_InputBuff & KEY_RIGHT ) work->sprt.X += 4;
    if( g_InputBuff & KEY_LEFT  ) work->sprt.X -= 4;

    //物体の移動
    work->LandMoveCount += LAND_MOVE_SPEED;
    work->Land.X = sin( work->LandMoveCount ) * 96 + (SCREEN_WIDTH / 2);
    //物体の大きさと座標情報を計算
    land_rect.left   = work->Land.X;
    land_rect.right = work->Land.X + LAND_WIDTH;
    land_rect.top    = work->Land.Y;
    land_rect.bottom= work->Land.Y + LAND_HEIGHT;
    //足元の座標を計算
    my_pos_x = work->sprt.X + HUMAN_WIDTH  / 2 ;
    my_pos_y = work->sprt.Y + HUMAN_HEIGHT + 1;

    //ジャンプ中は反応しない
    if( !work->JumpFlag )
    {//着地時の処理
        //キー入力でジャンプ処理
        if( g_DownInputBuff & KEY_Z     )
        {//ジャンプ開始時処理
            work->Acc = -JUMP_POWER;
            work->JumpFlag = true;
        }else{
            //足元をチェックし、もし何も無いのであれば落下を開始する
            if( !(my_pos_x > land_rect.left  && my_pos_x < land_rect.right  &&
                  my_pos_y > land_rect.top   && my_pos_y < land_rect.bottom ))
            {//落下処理開始
                work->Acc = 0;
                work->JumpFlag = true;
            }
        }
    }

    if( work->JumpFlag )
    {
```

```
        work->Acc += GRAVITY;
        work->sprt.Y += work->Acc;

        //段差への着地処理、まず下降中かをチェックする
        if( work->Acc > 0.0 )
        {//もし下降中に物体に接触したら、着地したとみなす
           if( my_pos_x > land_rect.left   && my_pos_x < land_rect.right  &&
               my_pos_y > land_rect.top    && my_pos_y < land_rect.bottom )
           {//物体上への着地処理
                //物体の座標を着地座標にする
                work->sprt.Y =  land_rect.top - HUMAN_HEIGHT;
                work->JumpFlag = false;
           }
        }

        //地面の着地処理
        if( work->sprt.Y > GRAND)
        {//座標が、地面座標より下ならば、着地したとみなす
            work->sprt.Y =  GRAND;
            work->JumpFlag = false;
        }
    }

    SpriteDraw( &work->sprt, 0);

    //物体の表示
    SpriteDraw( &work->Land, 1);

    g_pFont->DrawText( NULL,
        "Zキーでジャンプ、方向キーで移動",
        -1, &font_pos,
        DT_LEFT, 0xffffffff);
```

6-25 ジャンプの頂点位置を知る

ジャンプの高さを前もって知る

ここでは、ジャンプの頂点位置を知る方法を紹介します。

この処理は、使う機会はあまり多くないのですが、ゲーム上で、位置の予測を伴う処理の時などに必要となります。

予測とは言っても、手法の概念自体は単純で、要するにジャンプ位置の計算を、実際の処理をする前に、予め行なっておけばよいのです。

予測処理

では実際の処理を見ていきましょう。サンプルはＺキーでジャンプするときに、高さの予測を行ない表示するものです。

なおこのサンプルは、「6-17 ジャンプをする」を元に作成していますので、そちらも参照して下さい。

ここでは、処理のポイントだけを解説します。

予測処理のタイミング

プログラムは初期化後に、まずジャンプのチェックを行なって、ジャンプの処理を開始します。

ジャンプの高さの予測処理は、このジャンプ開始時の時に行なっています。

このタイミングであれば、ジャンプに必要な様々なデータがそろっているので、情報の取得がしやすくなるからです。

予測処理の概要

実際の予測処理ですが、これは、そのままジャンプの処理をまとめたものになっています。

すなわち、１ループ毎（１フレーム毎）に、重力値の加算を行ない、座標の高さを毎回計測しています。

この時、現在の位置が頂点位置かどうかを知るために、１つ前の位置を記録して比較を行なっています。

もし、高さが前回より低ければ落下が始まっているという事で、１つ前の位置が、頂点であるとみなす事ができます。

後は取得した頂点位置を記録して、表示等を行なっています。

LIST 6 －25－1 ジャンプの頂点位置を知る

```
void exec06_25(TCB* thisTCB)
{
#define JUMP_POWER  10.0  //ジャンプ力
#define GRAVITY  0.25     //重力値

    EX06_25_STRUCT* work = (EX06_25_STRUCT*)thisTCB->Work;
    float jump_height;
    float chk_height;
    float tmp_Acc;

    RECT font_pos = {  0,  0,640,480,};
    char str[128];

    //ジャンプ中は反応しない
    if( !work->JumpFlag )
    {//着地時の処理
        //キー入力でジャンプ処理
        if( g_DownInputBuff & KEY_Z    )
        {//ジャンプ開始時処理
            work->Acc = -JUMP_POWER;
            work->JumpFlag = true;

            //最高高度記録用に初期化
            work->JumpHeightMax = work->sprt.Y;

            tmp_Acc = work->Acc;
            jump_height = work->sprt.Y;
            //チェックの初期値はjump_heightより大きければOK
            chk_height = jump_height + 1.0;
            //高度のチェック
            //高度が下がり始めたらループの終了
            while(jump_height < chk_height)
            {//先立ってジャンプの計算を行なう
                chk_height = jump_height;
                tmp_Acc += GRAVITY;
                jump_height += tmp_Acc;
            }
            //表示用の変数に記録
            work->JumpHeight = chk_height;
```

```
        }
    }

    if( work->JumpFlag )
    {
        work->Acc += GRAVITY;
        work->sprt.Y += work->Acc;
        //ジャンプ中の最高高度を記録
        if(work->JumpHeightMax > work->sprt.Y)
            work->JumpHeightMax = work->sprt.Y;

        //着地したらジャンプ処理終了
        if( work->sprt.Y > SCREEN_HEIGHT / 2)
        {//座標が、地面座標より下ならば、着地したとみなす
            work->sprt.Y =  SCREEN_HEIGHT / 2;
            work->JumpFlag = false;
        }
    }

    SpriteDraw( &work->sprt, 0);

    //高さ表示
    sprintf( str,
        "予想高さ:%f¥n実測高さ:%f",
        work->JumpHeight,
        work->JumpHeightMax);

    g_pFont->DrawText( NULL, str, -1, &font_pos, DT_LEFT, 0xffffffff);

}
```

Chapter 7
ゲーム中の処理

7-1 キャラのアニメーション

アニメーションの原理

キャラクターのアニメーション処理を考えてみましょう。

アニメーションとゲームは切っても切れない間柄にあります。

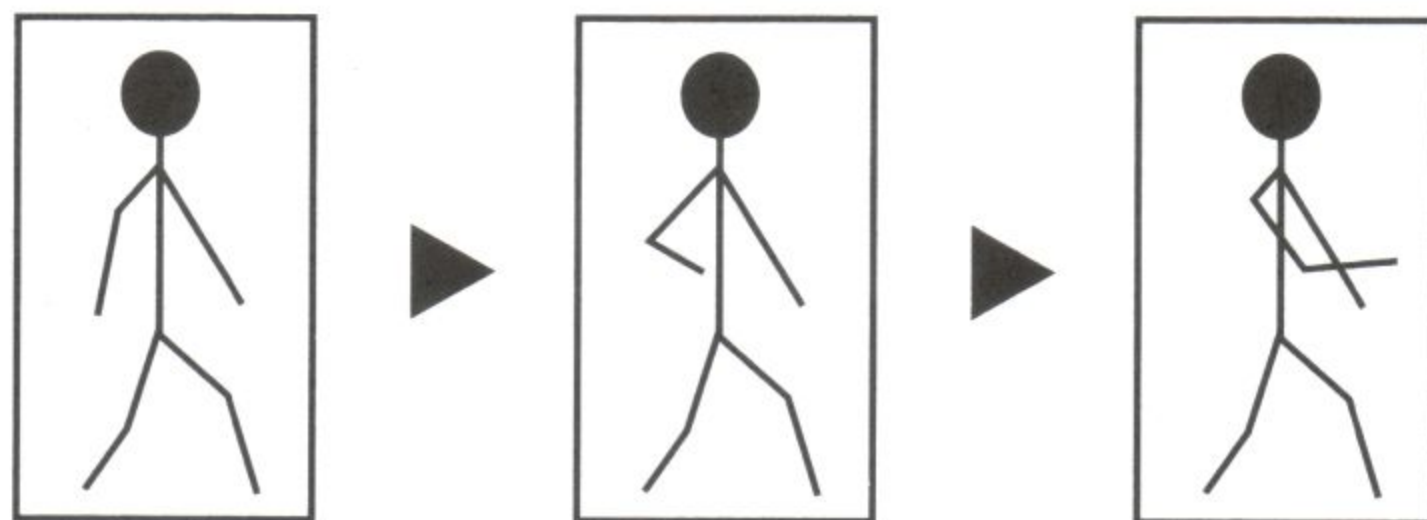
格闘ゲーム等はもとより、シューティングやRPG、最近ではアドベンチャーゲームまで何らかのアニメーションを行なっています。

実質的にアニメーション処理の無いゲームは存在しないと言っても良いでしょう。

では、ゲームで行なわれるアニメーションはどのような処理をしているのでしょうか。

答えを先に書くと、ゲームで行なわれる一般的なアニメーション処理はビットマップデータを切り替えることで行なわれます。

これは、テレビや映画などのアニメーションとまったく同じ事をプログラム上で行なっているのです。

図7 −1−1 アニメーション処理のイメージ図

少しずつ違う絵（ビットマップデータ）を連続で表示する事で動いて見える

アニメーションのプログラム

時間を管理

では実際の処理を見ていきましょう。

まず、アニメーションの時間を管理するために、アニメーション用のカウンタを用意します。

このカウンタに合わせて、表示するビットマップを切り替えます。

切り替えるタイミングはゲームや表示するキャラクターによって様々ですが、ここでは20フレームおきに切り替える様にします。

処理自体は単純で、毎フレームカウンターに時間を加算していき、時間が規定の値(20フレーム)に達したら切り替えの処理を行ないます。

画像の切り替え

さて実際の切り替え処理です。ビットマップを1枚ずつ切り替えてもいいのですが、アニメーションの枚数が増えると、データの管理が大変になってしまいます。

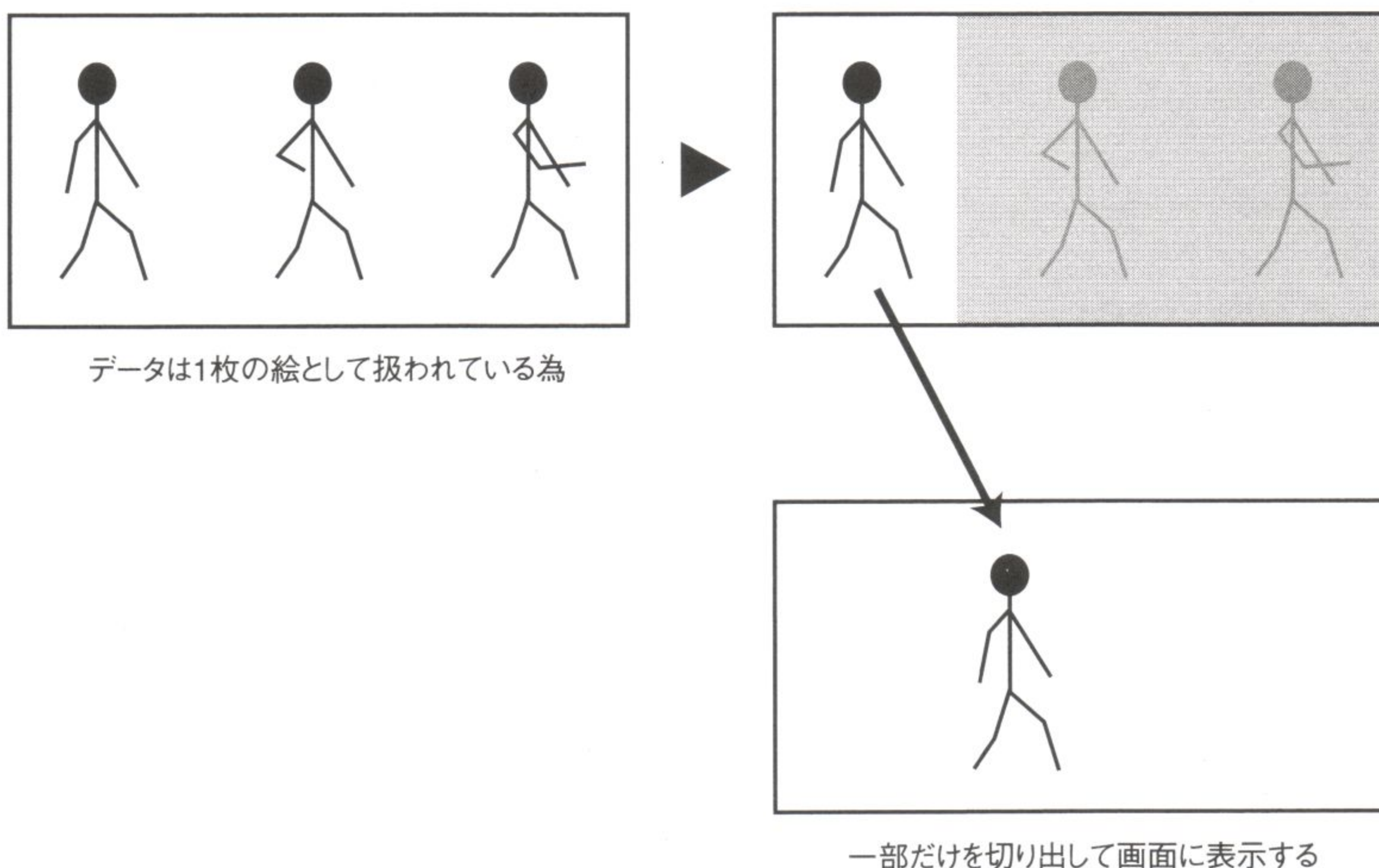
そこで、1枚のビットマップデータにアニメーションのパターンを用意しておき、その1部の領域を表示する事で、切り替えを行ないます。

一部領域の表示は、スプライトの機能にあります。

そこでサンプルでは、切り替え用のデータをRECT構造体で用意しておき、処理に合わせてスプライト処理に指定、パターンを切り替えるようにしています。

図7ー1ー2 一部を表示してアニメーションを行なう処理のイメージ図

データは1枚の絵として扱われている為

一部だけを切り出して画面に表示する

LIST 7ー1ー1 キャラのアニメーション

```
void init07_01(TCB* thisTCB)
{
    //使用するテクスチャの読み込み
    D3DXCreateTextureFromFile( g_pD3Ddevice, "..¥¥..¥¥data¥¥0033.png",&g_pTex[0] );
    D3DXCreateTextureFromFile( g_pD3Ddevice, "..¥¥..¥¥data¥¥0022.png",&g_pTex[1] );
}
```

```
void exec07_01(TCB* thisTCB)
{
#define ANIM_FRAME_MAX 4             //アニメーションのパターン数
#define ANIM_CHANGE_CNT 20           //アニメーションを切り替えるフレーム数

RECT  anim_table[] =
{ {  0,  0,  64, 128,},      //1枚目のアニメパターン
  { 64,  0, 128, 128,},      //2枚目のアニメパターン
  {128,  0, 192, 128,},      //3枚目のアニメパターン
  {192,  0, 256, 128,},      //4枚目のアニメパターン
};

    //タスクにワークを割り付ける
    SPRITE* pspr;
    pspr = (SPRITE*)thisTCB->Work;

    pspr->timer += 1.0;

    if(pspr->timer >= ANIM_CHANGE_CNT)
    {
       //タイマーを戻す
       pspr->timer -= ANIM_CHANGE_CNT;
       //表示パターンを1パターン進める
       pspr->Frame++;
       if( pspr->Frame >= ANIM_FRAME_MAX )
       {//表示パターンを最初に巻き戻す
           pspr->Frame = 0;
       }
    }

    pspr->SrcRect = &anim_table[pspr->Frame];
    pspr->X = 320;
    pspr->Y = 240;
    SpriteDraw(pspr,0);
}
```

7-2 キャラクターを動かす

キー入力で自機を動かす

主人公となるキャラクターを動かしてみましょう。

ここでは、方向キーにより８方向にキャラクターを動かしてみます。

自機を動かす基本的なプログラムで、入力操作のの基本となるものです

基本的にさほど難しい事はなく、キー入力の状態を判別し、それに合わせて表示座標を書き換えてやります。

ただ、少し問題となるのは斜め方向への移動です。そのままですと移動速度が斜めだけ速くなってしまうので、[6－1]で紹介した手法を使い、一定速度で動くようにしています。

LIST 7－2－1 キャラクターを動かす

```
void init07_02(TCB* thisTCB)
{
    SPRITE* pspr;
    pspr = (SPRITE*)thisTCB->Work;//ワークを割り当てる

    //使用するテクスチャの読み込み
    D3DXCreateTextureFromFile( g_pD3Ddevice, "..¥¥..¥¥data¥¥0044.png",&g_pTex[0] );

    //初期座標
    pspr->X = 320;
    pspr->Y = 240;
}
void exec07_02(TCB* thisTCB)
{
//移動速度
#define MOVE_SPEED 2.0f

    SPRITE* pspr;
    float   direction;

    pspr = (SPRITE*)thisTCB->Work;//ワークを割り当てる

```

```
    switch( g_InputBuff & 0x0f )  //移動キーのみを抽出
    {//移動方向を算出する
        case 0x01: direction= atan2( -1.0,  0.0); break; //上
        case 0x02: direction= atan2(  1.0,  0.0); break; //下
        case 0x04: direction= atan2(  0.0, -1.0); break; //左
        case 0x08: direction= atan2(  0.0,  1.0); break; //右

        case 0x05: direction= atan2( -1.0, -1.0); break; //左上
        case 0x06: direction= atan2(  1.0, -1.0); break; //左下
        case 0x09: direction= atan2( -1.0,  1.0); break; //右上
        case 0x0A: direction= atan2(  1.0,  1.0); break; //右下

        default:  SpriteDraw(pspr,0); return;
    }
    //移動方向から移動値を取得し実際の移動処理を行なう
    pspr->X += cos(direction)*MOVE_SPEED;
    pspr->Y += sin(direction)*MOVE_SPEED;

    SpriteDraw(pspr,0);
}
```

7-3 画面内での移動

動ける場所を制限する

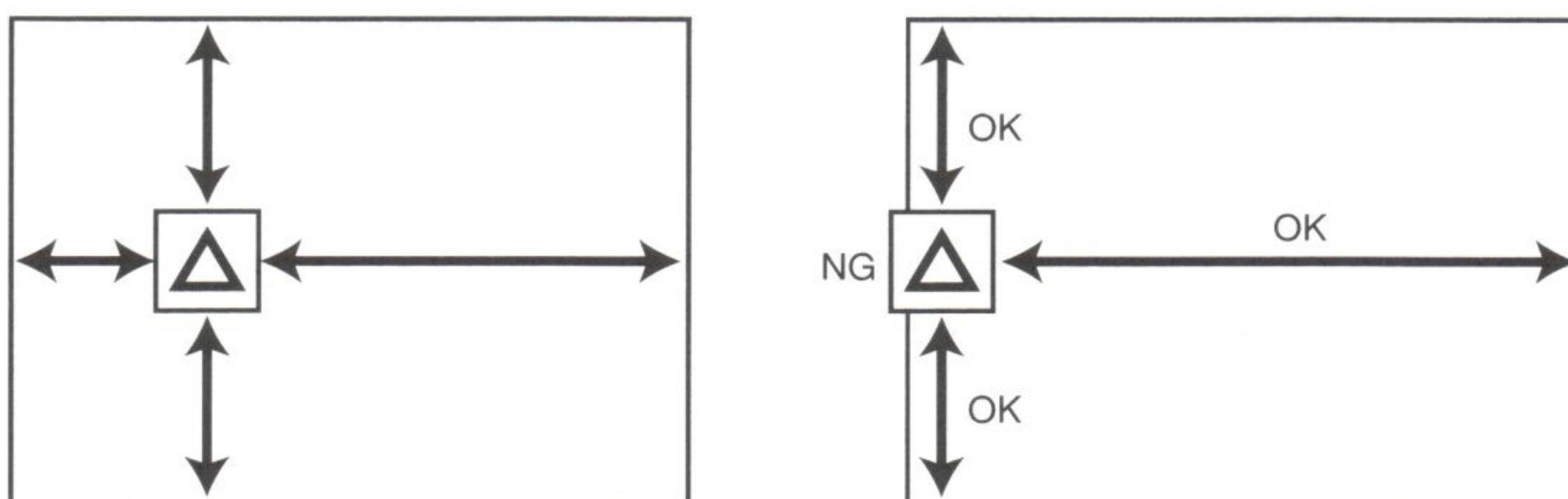
キャラクターを移動させる際、画面外に移動してしまうのは、ゲーム上も操作性上もあまり良い事ではありません。

そこで、ここでは画面外へ移動せず、画面内でのみ移動できるようにしてみます。

その仕組みですが、まず移動させるキャラクターの大きさデータを用意します。

そのデータを、画面の上下左右の座標と比較し、もし超えていたら、それ以上移動しないようにしてやります。

図7－3－1 大きさデータを画面枠と比較するイメージ図

OK
OK
NG
OK

判定枠を画面の枠と比較し
画面外に出ていたら、補正してやる

プログラムの解説

実際のプログラムでも同様ですが、比較の際に計算しやすいよう、いったん計算結果をRECT構造体に格納しています。

あと、この処理は数が少ない時は良いのですが、数が増えると比較的重めの処理になります。

そのため、多数のキャラクターで使用するときは注意が必要です。

LIST 7－3－1 画面内での移動

```
void init07_03(TCB* thisTCB)
{
    SPRITE* pspr;
```

```
    pspr = (SPRITE*)thisTCB->Work;//ワークを割り当てる

    //使用するテクスチャの読み込み
    D3DXCreateTextureFromFile( g_pD3Ddevice, "..¥¥..¥¥data¥¥0044.png",&g_pTex[0] );

    //初期座標
    pspr->X = 320;
    pspr->Y = 240;
}
void exec07_03(TCB* thisTCB)
{
//移動速度
#define MOVE_SPEED 8.0f

    SPRITE* pspr;
    float   direction;
    RECT size_rect = { 0, 0, 64, 64, };
    RECT hit_rect;

    pspr = (SPRITE*)thisTCB->Work;//ワークを割り当てる

    switch( g_InputBuff & 0x0f )  //移動キーのみを抽出
    {
        case 0x01: direction= atan2( -1.0,  0.0); break; //上
        case 0x02: direction= atan2(  1.0,  0.0); break; //下
        case 0x04: direction= atan2(  0.0, -1.0); break; //左
        case 0x08: direction= atan2(  0.0,  1.0); break; //右

        case 0x05: direction= atan2( -1.0, -1.0); break; //左上
        case 0x06: direction= atan2(  1.0, -1.0); break; //左下
        case 0x09: direction= atan2( -1.0,  1.0); break; //右上
        case 0x0A: direction= atan2(  1.0,  1.0); break; //右下

        default:  SpriteDraw(pspr,0); return;
    }
    //移動処理
    pspr->X += cos(direction)*MOVE_SPEED;
    pspr->Y += sin(direction)*MOVE_SPEED;

    //画面外判定チェックの為、キャラの大きさの矩形を計算し
```

```
    //一旦構造体に格納する
    hit_rect.top    =   pspr->Y + size_rect.top;
    hit_rect.bottom =   pspr->Y + size_rect.bottom;
    hit_rect.left   =   pspr->X + size_rect.left;
    hit_rect.right  =   pspr->X + size_rect.right;

    //上下左右、それぞれの方向に対して画面外かどうかをチェックし
    //それ以上移動しないようにする
    if(hit_rect.left   < 0)
    {//左
        pspr->X = 0;
    }
    if(hit_rect.top    < 0)
    {//上
        pspr->Y = 0;
    }

    if(hit_rect.right  > SCREEN_WIDTH)
    {//右
        pspr->X = SCREEN_WIDTH - size_rect.right;
    }
    if(hit_rect.bottom > SCREEN_HEIGHT)
    {//下
        pspr->Y = SCREEN_HEIGHT - size_rect.bottom;
    }

    SpriteDraw(pspr,0);
}
```

7-4 複数同時プレイ

複数同時プレイを実現するには

複数のキャラクターを同時に動かす、複数同時プレイを考えてみましょう。

対戦格闘ゲームではもちろん、最近ではシューティングゲームでも2人同時プレイが可能なものが多くあります。

また、アクションゲーム等の中には2人だけではなく3人以上のキャラクターを、同時に操作するものもあります。

こういった処理を行なう場合、1つ1つの操作に対して個別に作成してはとても大変です。

そこで一般にはプレイヤーIDを導入し、それに応じて処理を行なう事になります。

入力処理はプレイヤーIDに応じた値を返すようにし、処理側はどのプレイヤーで処理をしても良いように作成すれば、非常にシンプルに処理を作成する事が出来ます。

複数同時プレイのプログラム

初期化

ではサンプルプログラムを見ていきます。

内容は2機の自機を、キーボードの方向キーと、ジョイステック方向キーで別々に操作するものです。

自機の移動は並列動作のため、タスクを作成して処理しています。

まずは初期化です。自機の移動用のタスクexec07_04_myshipを作成しています。

見てわかるとおり、同じタスク処理関数でを2つのプレイヤーを管理しており、初期座標を除けば、唯一の違いはプレイヤーIDだけです。

タスク処理

実際のタスク処理の内部もシンプルで、入力関数EX07_04_player_inputにプレイヤーIDを渡して、帰ってきた入力値を元に自機を移動させるだけとなっています。

この処理の大事な事は、基本的な共通部分はそのままプログラムし、プレイヤーによって変わる部分はIDを引数とした関数にする事で、複雑な部分を隠蔽しているという事です。

ここでは入力による移動だけですが、他の処理もこの方針でプログラムすれば、多人数プレイに対応する事が出来ると思います。

なおサンプルでは2人同時プレイですが、IDを増やして入力関数を対応させれば、そのまま何人でも対応が可能です。

最後に入力関数ですが、これは単純にプレイヤーIDを見て入力を返す処理です。

プレイヤー識別はswitch文で行なっており、内容に応じて入力取得をする変数を切り替えています。

LIST 7－4－1 複数同時プレイ

```
typedef struct{
    SPRITE          MyShip;
    int             PID;
} EX07_04_STRUCT;


#define MOVE_SPEED     4.0
#define PLAYER_1       0
#define PLAYER_2       1
#define PLAYER_3       2
#define PLAYER_4       3



unsigned char EX07_04_player_input( int PlayerID )
{    unsigned char input_key;

    //入力値をプレイヤーによって変更する
    switch( PlayerID )
    {//キーボードをプレイヤー1、ジョイスティックをプレイヤー2とする
        case PLAYER_1: input_key = g_KeyInputBuff; break;
        case PLAYER_2: input_key = g_JoystickBuff; break;
    }

    return input_key;
}

void exec07_04_myship(TCB* thisTCB)
{
    EX07_04_STRUCT* work = (EX07_04_STRUCT*)thisTCB->Work;
    unsigned char input_buff;

    //プレイヤーIDによる入力
    input_buff = EX07_04_player_input( work->PID );

```

```
        //キー入力による移動
        if( input_buff & KEY_UP    ) work->MyShip.Y -= MOVE_SPEED;
        if( input_buff & KEY_DOWN  ) work->MyShip.Y += MOVE_SPEED;
        if( input_buff & KEY_RIGHT ) work->MyShip.X += MOVE_SPEED;
        if( input_buff & KEY_LEFT  ) work->MyShip.X -= MOVE_SPEED;

        SpriteDraw( &work->MyShip,0);
    }

    void init07_04(TCB* thisTCB)
    {
        TCB*    myship_tcb;
        EX07_04_STRUCT* myship_work;
        int loop;

        //使用するテクスチャの読み込み
        D3DXCreateTextureFromFile( g_pD3Ddevice, "..¥¥..¥¥data¥¥0044.png",&g_pTex[0] );

        //自機を2機作成する
        for( loop = 0;loop < 2; loop++ )
        {
            myship_tcb = TaskMake( exec07_04_myship, 0x2000 );
            myship_work = (EX07_04_STRUCT*)myship_tcb->Work;
            //プレイヤーにより初期表示座標を変える
            myship_work->MyShip.X =
                ((SCREEN_WIDTH  / 2) - 64) + loop * 128;
            myship_work->MyShip.Y =
                SCREEN_HEIGHT / 2;
            //プレイヤーIDを登録
            myship_work->PID = loop;
        }
    }

    void exec07_04(TCB* thisTCB)
    {
        //複数同時処理のため、サンプルではメイン処理は何もしない
    }
```

7-5 キャラクターの機数を管理する

自機の残機管理

機数の管理処理はシンプルですが、ゲームにおいては非常に重要な処理です。

通常、ゲーム開始時の残機数は一定ですが、ゲーム中の行動、例えば高得点を取得したり、アイテムを取得するなどの行為によって残機数が増えます。

これによりゲームのプレイ時間が決まるため、プレイヤーの感じる体感的な難易度や、場合によっては最終的なゲームバランスまで影響します。

そのため、残りの機数により難易度が変化するというゲームも存在します。

ここでは、簡単な機数管理の例として、スコアによる機数アップの処理と、機数が無くなった時のゲームオーバーの処理を紹介します。

残機を管理する処理の概要

サンプルプログラムの概要ですが、Zキーでスコアを100点ずつ加算し、Xキーで残り機数を減らします。

スコアは1000点を越える毎に機数が一機ずつ増えていきます。残機数が0以下になったらしゲームオーバーのメッセージを表示します。

では実際の処理を見ていきます。

残機管理のプログラム

初期化

まず、初期化処理では、残機数と最初に機数が増えるスコアを設定します。

次にメインの処理ですが、まず、キー入力に応じてスコア加算と機数の減算処理を行ないます。

スコア加算

スコア加算は単純に加算のみですので問題は無いと思いますが、機数を減らす処理では、ゲームオーバーの処理も同時に行なっています。

ここではキー入力をみて処理を行なっていますが、実際には当たり判定の処理等を施した上で処理する事になるでしょう。

機数増加

次は、スコアによる機数の増加処理です。

これは、機数が増えるスコアを最初に登録しておき、そのスコアを越えたら、次に増えるスコアを再度登録する、というアルゴリズムです。

この処理アルゴリズムですと、一定点数毎に自機が増えたり、一定機数増えたらそれ以上増えないようする、という事が簡単に出来ます。

処理の最後は、スコア及び残機数表示と、ゲームオーバーメッセージの処理を行なっています。

LIST 7－5－1 キャラクタの機数を管理する

```
#define DEFAULT_SHIP    3
#define BONUS_SCORE     1000

typedef struct{
    int             Score;
    int             MyShipCount;
    int             NextScore;      //次に残機が増えるスコア
    int             GameOver;
} EX07_05_STRUCT;


void init07_05(TCB* thisTCB)
{
    EX07_05_STRUCT* work = (EX07_05_STRUCT*)thisTCB->Work;

    //残機数と、ボーナススコアの初期化
    work->MyShipCount = DEFAULT_SHIP;
    work->NextScore = BONUS_SCORE;
}

void exec07_05(TCB* thisTCB)
{
    EX07_05_STRUCT* work = (EX07_05_STRUCT*)thisTCB->Work;
    RECT font_pos = {  0,  0,640,480,};
    char str[128];

    //Zキーでスコアを加算
```

```
	if( g_DownInputBuff & KEY_Z ) work->Score += 100;
	//Xキーで残機を1機減らす
	if( g_DownInputBuff & KEY_X )
	{
		work->MyShipCount--;
		if( work->MyShipCount < 0 )
		{//ゲームオーバー処理
			work->GameOver = true;
		}
	}

	//スコアによる残機増加処理
	if( work->Score >= work->NextScore )
	{
		work->MyShipCount++;
		work->NextScore += BONUS_SCORE;
	}

	//スコアと、機数の表示
	sprintf( str,"SCORE: %8d  残り機数:%2d¥n次のボーナススコア:%8d",work->Score,work->MyShipCount,work->NextScore);
	g_pFont->DrawText( NULL, str, -1, &font_pos, DT_LEFT, 0xffffffff);

	if( work->GameOver )
	{//ゲームオーバー時にメッセージ表示
		font_pos.top += 48;
		g_pFont->DrawText( NULL, "GAME  OVER!!", -1, &font_pos, DT_LEFT, 0xffffffff);
	}
}
```

7-6 メニューの表示

ゲームのメニュー画面

メニューの表示はゲームのインターフェースとして最もポピュラーなものです。

しかし、その実装や表示手法は様々で、メニューとしてひとくくりに説明する事は非常に難しいところです。

そこで、ここでは最も一般的と思われる、「選択マークによるメニューの選択と決定」の処理を解説していきます。

完全ではありませんが、これを基本とすれば、その他のメニュー表示にも応用が利くでしょう。

メニュー画面の構造

では、概要から説明していきます。

まず、メニューのデータ構造からです。基本的にメニューは、「メニュー項目」と「メニューの実行」から構成されています。

そのため、この2つの項目をペアでデータ内に格納する事が基本となります。

そして、このメニューの項目を表示する時、同時に選択マークを表示してやり、選択中の項目が決定されたら、ペアで格納されている処理を実行してやります。

基本的には以上の処理になりますが、サンプルではこれにメニューの詳細解説データを加え、選択時に同時に表示するようにしています。

メニュー画面プログラム

処理概要

では実際のプログラムをみていきましょう。サンプルは方向キーでメニューを選択し、Zキーでメニューを実行するものです。

なお、メニュー表示時のレイアウト処理のサンプルとして、Xキーを押しながら方向キーでメニューの移動を行なえるようにしています。

まずメニューデータですが、先に示したように、実行する関数と表示するメッセージで構築されています。

メニューの数は全部で5種類で、NULLの関数は終了を意味します。

メニューの移動選択処理

次に実際の処理です。

まず、メニューの移動処理を行なっています。これは、単純に表示座標を増減させるだけですので問題は無いでしょう。

次に、メニューの選択処理です。キー入力により選択中のメニューの項目を示す変数SelectMenuの値を増減させています。

この時メニューの項目数を越えないように比較、補正しています。もしメニューの種類が複数の場合は、この項目数もデータに加えて処理するようにすると良いでしょう。

メニュー実行処理

そして次に、メニューの実行処理を行ないます。選択中のメニューデータに格納されている処理関数を実行してやります。

ここでは、四則演算を行なう関数を4種類用意し、メニューによってそれぞれを実行するようにしています。

この時、関数が無く、NULLポインタであれば、メニューの終了処理を行ないます。

次に、メニューの表示処理です。

選択マークの表示

まず、選択マークの位置を計算します。

マークはメニューの横につくため、X座標はそのまま、Y座標は選択中のメニュー項目の高さに合わせてやります。

そして、メニュー項目を表示します。メニューデータ全てのループを行ない、メニュー項目の内容を全部表示してやります。

メニューの詳細表示

最後に、選択中のメニューの詳細表示を行ないます。

これは、選択中のメニューデータの項目から詳細解説用のデータを表示してやるだけです。

このサンプルではメニューは単純な表示に留めていますが、色々工夫して凝った表示にしたり、メニューデータにに付加するデータを増やして改良してみても良いでしょう。

LIST 7 －6－1 メニューの表示

```
#define MENU_INDEX  5
//メニューメッセージの高さ
#define MENU_HEIGHT  16

```

```
typedef struct{
    int (*MenuFunc)(int);               //メニュー実行関数
    char*           MenuName;           //メニュー名
    char*           HelpMess;           //メニュー選択時に表示するヘルプメッセージ
} EX07_06_MENU_DATA;


typedef struct{
    SPRITE          SelectMark;
    int             SelectMenu;
    char            DispMess[64];
    int             Num;
    int             MenuX;
    int             MenuY;


} EX07_06_STRUCT;


int EX07_06_add( int num )
{
    return num + 100;
}
int EX07_06_sub( int num )
{
    return num - 100;
}
int EX07_06_mult( int num )
{
    return num * 2;
}
int EX07_06_div( int num )
{
    return num / 2;
}


void init07_06(TCB* thisTCB)
{
    //使用するテクスチャの読み込み
```

```
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥select_mark.png",&g_pTex[0] );
}

void exec07_06(TCB* thisTCB)
{
    EX07_06_STRUCT* work = (EX07_06_STRUCT*)thisTCB->Work;
    int loop;
    RECT font_pos = { 0, 0, 640, 480, };
    char str[128];

    EX07_06_MENU_DATA menu_data[] =
{
    {//メニュー1
      EX07_06_add,
      "メニュー1:加算",
      "サンプル用メニュー1です。値を100加算します。",
    },{//メニュー2
      EX07_06_sub,
      "メニュー2:減算",
      "サンプル用メニュー2です。100減算します。",
    },{//メニュー3
      EX07_06_mult,
      "メニュー3:乗算",
      "3つ目のメニューです。値を2倍します。",
    },{//メニュー4
      EX07_06_div,
      "メニュー4:除算",
      "4つ目の選択メニュー。値を半分にします。",
    },{//メニュー5
      NULL,
      "終了",
      "関数ポインタをNULLにして、終了扱いにします。(終了処理は実装していません)",
    },
};


    if( g_InputBuff & KEY_X  )
    {//Xキーを押しながらだと、メニューの移動
        if( g_InputBuff & KEY_UP    ) work->MenuY -= 4;
```

```
        if( g_InputBuff & KEY_DOWN  ) work->MenuY += 4;
        if( g_InputBuff & KEY_RIGHT ) work->MenuX += 4;
        if( g_InputBuff & KEY_LEFT  ) work->MenuX -= 4;
    } else {
      //押していない時はメニューの選択
        if( g_DownInputBuff & KEY_DOWN )
        {
            work->SelectMenu++;
            //メニュー数チェック
            if( work->SelectMenu >= MENU_INDEX )work->SelectMenu = 0;
        }
        if( g_DownInputBuff & KEY_UP  )
        {
            work->SelectMenu--;
            if( work->SelectMenu < 0 ) work->SelectMenu = MENU_INDEX-1;
        }

        if( g_DownInputBuff & KEY_Z  )
        {//Zキーで選択中のメニューの実行
            if( menu_data[ work->SelectMenu ].MenuFunc == NULL )
            {//ただし、実行する関数が無い場合は終了
                //メニュー終了処理
            }else{
                //メニュー内容の実行
                work->Num = menu_data[ work->SelectMenu ].MenuFunc( work-
>Num );
            }
        }
    }

    //選択マークの位置を決定
    work->SelectMark.X = work->MenuX;
    work->SelectMark.Y = work->MenuY + work->SelectMenu * MENU_HEIGHT;
    //選択マークの表示
    SpriteDraw(&work->SelectMark,0);

    //メニューの表示
    font_pos.top  = work->MenuY;
    font_pos.left = work->MenuX + MENU_HEIGHT;
    for(loop = 0; loop < MENU_INDEX; loop++)
```

```
    {
        g_pFont->DrawText( NULL, menu_data[ loop ].MenuName, -1, &font_pos,
DT_LEFT, 0xffffffff);
        font_pos.top += MENU_HEIGHT;
    }

    //下段の固定位置にメニューによる数値変化とメニュー詳細内容の表示
    font_pos.top  = 448;
    font_pos.left = 32;
    sprintf( str,"現在の数値： %d¥n%s",work->Num, menu_data[ work->SelectMenu
].HelpMess);
    g_pFont->DrawText( NULL, str, -1, &font_pos, DT_LEFT, 0xffffffff);
}
```

7-7 キャラなどの進行方向の向きを知る

進行方向を判別する

移動する物体の進行方向を割り出す処理を作ってみましょう。

この処理は相手の移動先を予測したり、相手が今何処を向いているかを知る事が出来ます。

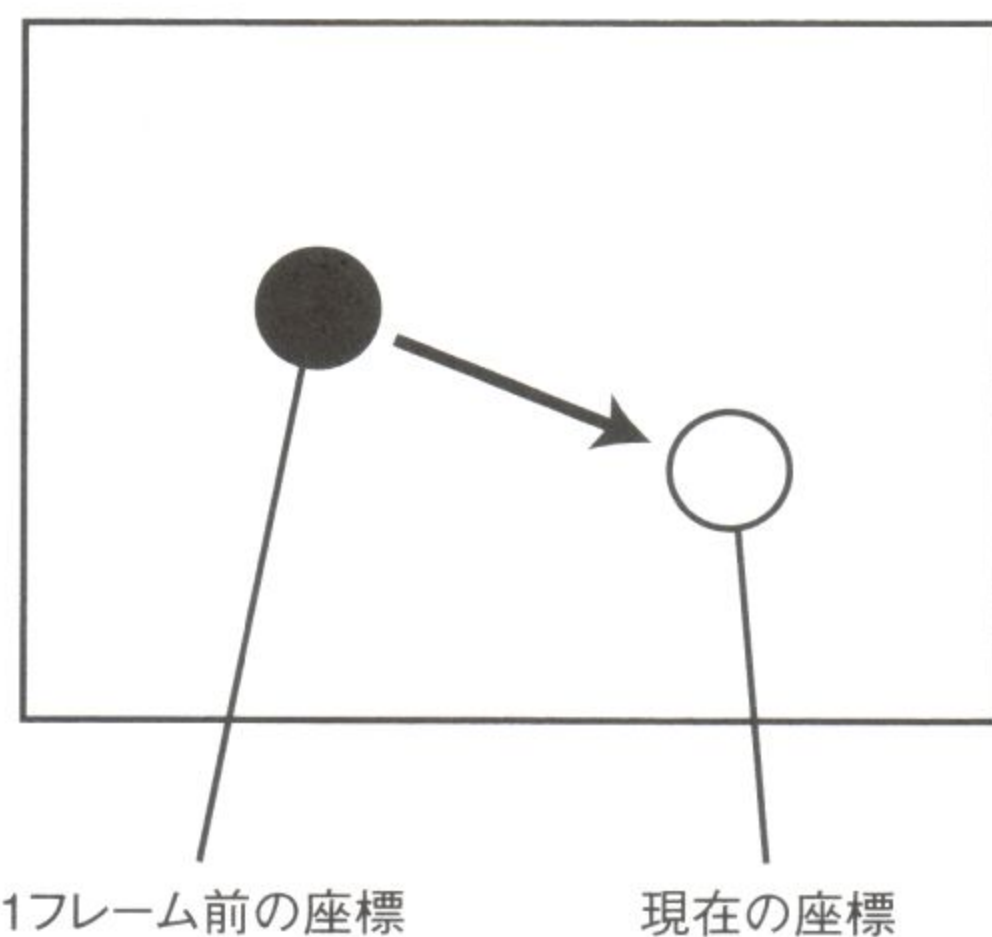
判別処理の概要

その処理方法ですが、まず、現在の座標を記録しておきます。

そして次のフレームで移動を行なったら、記録していた座標と、移動先の座標の相対位置を算出し、数学関数atan2を用いて、方向を計算します。

この処理方法の利点は、内部に方向を管理するデータを持たなくても良い点です。

ただ、処理の手順によって前フレームの座標を保持する必要がありますが、その場合でも逆に座標さえ保持しておけば、方向を割り出す事が出来ます。

図7－7－1 フレームを記録しておき、移動先との位置で方向を割り出すイメージ図

進行方向を判別するプログラム

では実際のプログラムを見ていきます。サンプルは自機を動かし、その方向を表示するプログラムです。

まず、初期処理後、現在の座標を保持し移動の処理を行ないます。サンプルでは入力に合わせて自機を8方向に動かしています。

次に、移動後の座標から、相対位置を計算し、atan2関数を使用して方向を計算します。

この時、座標が移動していないと方向が計算できないため、移動していない場合に0度の方向を示すよう特別扱いをしています。

最後に座標の表示と自機のスプライトの表示を行ない、処理は終了です。

サンプルでは8方向移動のため、表示される値は8つだけですが、プログラム自体は多方向に対応しているので、実際に使用する際でも大きく変更する事は無いと思います。

LIST 7－7－1 進行方向の向きを知る

```
void init07_07(TCB* thisTCB)
{
    //使用するテクスチャの読み込み
    D3DXCreateTextureFromFile( g_pD3Ddevice, "..¥¥..¥¥data¥¥0044.png",&g_pTex[0] );
    D3DXCreateTextureFromFile( g_pD3Ddevice, "..¥¥..¥¥data¥¥0055.png",&g_pTex[1] );
}

void exec07_07(TCB* thisTCB)
{
#define MOVE_SPEED  8.0
    SPRITE* work = (SPRITE*)thisTCB->Work;
    float beforeX;
    float beforeY;
    float posX;
    float posY;
    float direction;
    char str[128];
    RECT font_pos = { 0, 0, 640, 480, };

    //移動前の座標を保存
    beforeX = work->X;
    beforeY = work->Y;

    //キー入力による移動
    if( g_InputBuff & KEY_UP    ) work->Y -= MOVE_SPEED;
    if( g_InputBuff & KEY_DOWN  ) work->Y += MOVE_SPEED;
    if( g_InputBuff & KEY_RIGHT ) work->X += MOVE_SPEED;
    if( g_InputBuff & KEY_LEFT  ) work->X -= MOVE_SPEED;
```

```
    //移動前の座標と現在の座標との相対座標を計算
    posX = work->X - beforeX;
    posY = work->Y - beforeY;

    //相対座標から方向を計算
    if( posX == 0 && posY == 0 )
    {//座標が移動していなかった時は方向が定まらないため、特別扱い
        direction = 0.0;
    }else{
        direction = atan2( posY , posX );
        direction = -direction / M_PI * 180.0;
    }

    //方向の表示
    sprintf( str,"方向　%f",direction);
    g_pFont->DrawText( NULL, str, -1, &font_pos, DT_LEFT, 0xffffffff);

    SpriteDraw( work, 0);
}
```

7-8 目標を狙って弾を撃たせる

自機を狙って弾を撃つ

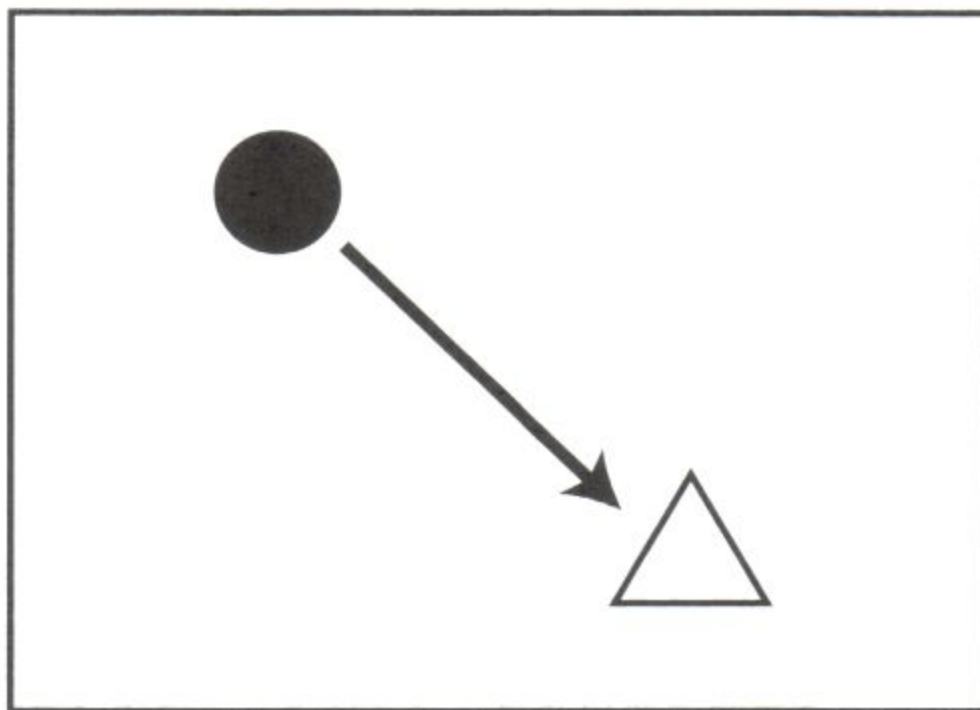
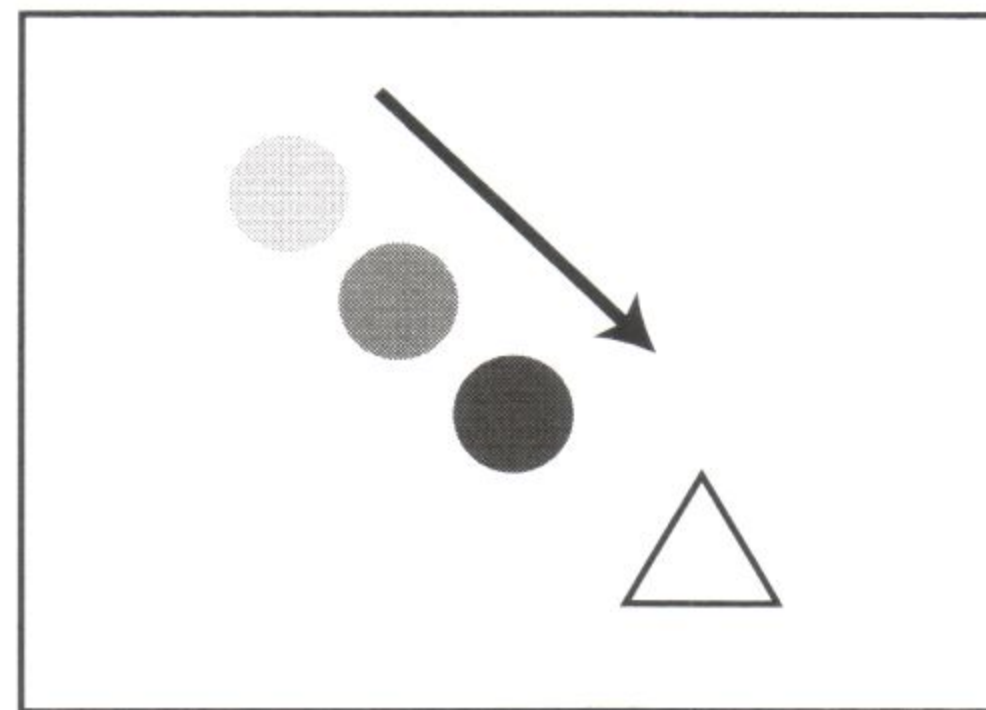
自機を狙って、弾を発射させる処理を作ってみましょう。

弾の発射処理はシューティングゲーム等では基本ともいえる処理で、この処理が無いとゲーム作りは不可能といってもいいでしょう。

狙って撃つ処理の概要

それでは、概念から解説していきましょう。

相手を狙う弾の発射は大きく2つの処理に分けられます。1つは相手の方向を知る処理、もう1つは相手に向かって弾を進める処理です。

図7－8－1 相手を狙って弾を撃つ処理は2つに分けられるイメージ図

1つは、相手の方向を割り出す処理
（数学関数atan2を使用）

もう一つは、その方向に向けて弾が進む処理

相手を狙う

まず、相手を狙う処理ですが、こちらは数学関数atan2を使用します。

この関数に、自分に対する相手の位置を引数にして呼び出すことで、相手の方向がラジアン単位で返ってきます。

弾を移動させる

次に、相手に向かって弾を進める処理ですが、こちらはcos関数とsin関数を使用します。

この関数に、相手の方向を引数として渡すと、弾を進めるのに必要な座標の増分値が返ってきます。

X方向の増分値がcos関数で、Y方向の増分値がsin関数です。

弾を撃つプログラム

弾の初期化

それでは、プログラムを見ていきしょう。サンプルでは、画面中央から自機を狙って、一定時間毎に弾が発射されます。

はじめにメイン側の処理ですが、初期化後にTaskMakeで弾を発射する処理を作成しており、その後自機の移動だけを行なってます。

肝心の弾の発射処理ですが、まず座標の初期化を行ないます。そして、相手を狙うため、自分と相手の座標を取得します。

次に、座標から相手がどの方向にいるかを計算し、自分の座標を基準とした相手との位置を引数として、計算関数atan2に渡します。

atan2の引数は、X、Yの順ではなく、Y、Xなので注意してください。

atan2から得られた方向から、X、Yの増分値を計算します。これはそのまま、sin、cos関数に渡すだけです。

得られた値はそのままだと、1フレーム当たり1.0しか進みませんので移動スピードを掛けてやります。

以上で、弾の初期化処理は終了です。

メイン処理

弾のメイン処理ですが、いたってシンプルで、初期化で得られた増分値を毎フレーム加算してやるだけです。

処理がシンプルなので、弾を大量に出す場合でもそれほど処理に負荷がかかる事は無いでしょう。

LIST 7 −8−1 弾を発射する

```
typedef struct{
    SPRITE          sprt;
    SPRITE*         Target;                 //目標のスプライト
    int             Time;
    float           AddX;                   //増分値X
    float           AddY;                   //増分値Y
} EX07_08_STRUCT ;

void exec07_08_bullet(TCB* thisTCB)
{
```

```
#define MOVE_SPEED  24.0
    EX07_08_STRUCT* work = (EX07_08_STRUCT*)thisTCB->Work;
    //自弾の座標
    float my_X;
    float my_Y;
    //目標の座標
    float targetX;
    float targetY;
    //目標の方向
    float direction;

    //初期化処理、一定時間ごとに初期化される
    if( work->Time++ == 30 )
    {//一定時間ごとに座標と増分値を計算
        work->sprt.X = SCREEN_WIDTH  / 2;
        work->sprt.Y = SCREEN_HEIGHT / 2;
        work->Time = 0;

        //弾の座標を取得
        my_X     = work->sprt.X;
        my_Y     = work->sprt.Y;
        //目標の自機座標を取得
        targetX = work->Target->X + 16;
        targetY = work->Target->Y + 16;

        //座標から相手への方向を計算
        direction = atan2( targetY - my_Y, targetX - my_X );
        //方向から、X、Yそれぞれの座標増分値を計算
        work->AddX = cos( direction ) * MOVE_SPEED;
        work->AddY = sin( direction ) * MOVE_SPEED;
    }

    //座標に増分値を加算
    work->sprt.X += work->AddX;
    work->sprt.Y += work->AddY;

    SpriteDraw(&work->sprt,1);
}

void init07_08(TCB* thisTCB)
```

```
{
    TCB*     tmp_tcb;
    SPRITE* sprt;
    EX07_08_STRUCT* tmp_work;

    //使用するテクスチャの読み込み
    D3DXCreateTextureFromFile( g_pD3Ddevice, "..¥¥..¥¥data¥¥0044.png",&g_pTex[0] );
    D3DXCreateTextureFromFile( g_pD3Ddevice, "..¥¥..¥¥data¥¥0055.png",&g_pTex[1] );

    //自機のスプライト
    sprt = (SPRITE*)thisTCB->Work;
    sprt->X = SCREEN_WIDTH  / 2;
    sprt->Y = SCREEN_HEIGHT / 2 + 120;

    //初期化と、目標スプライトを設定
    tmp_tcb = TaskMake( exec07_08_bullet, 0x2000 );
    tmp_work = (EX07_08_STRUCT*)tmp_tcb->Work;
    tmp_work->Target = sprt;

}

void exec07_08(TCB* thisTCB)
{
#define MOVE_SPEED  8.0
    SPRITE* work = (SPRITE*)thisTCB->Work;

    //キー入力による移動
    if( g_InputBuff & KEY_UP    ) work->Y -= MOVE_SPEED;
    if( g_InputBuff & KEY_DOWN  ) work->Y += MOVE_SPEED;
    if( g_InputBuff & KEY_RIGHT ) work->X += MOVE_SPEED;
    if( g_InputBuff & KEY_LEFT  ) work->X -= MOVE_SPEED;

    SpriteDraw( work, 0);
}
```

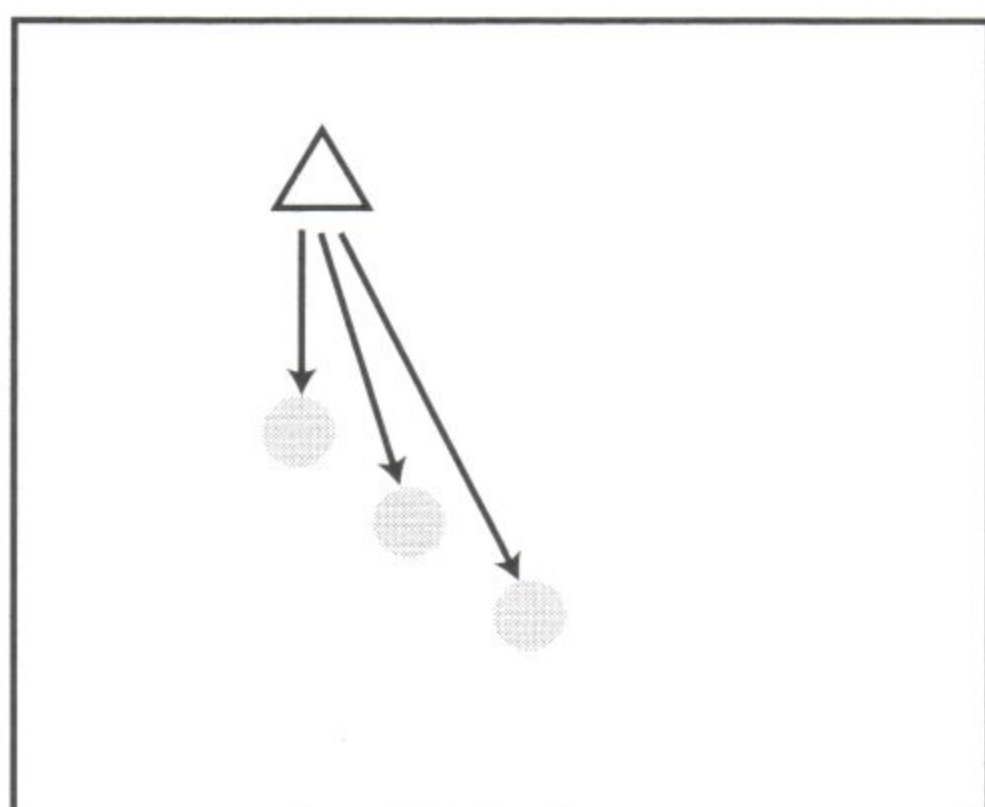
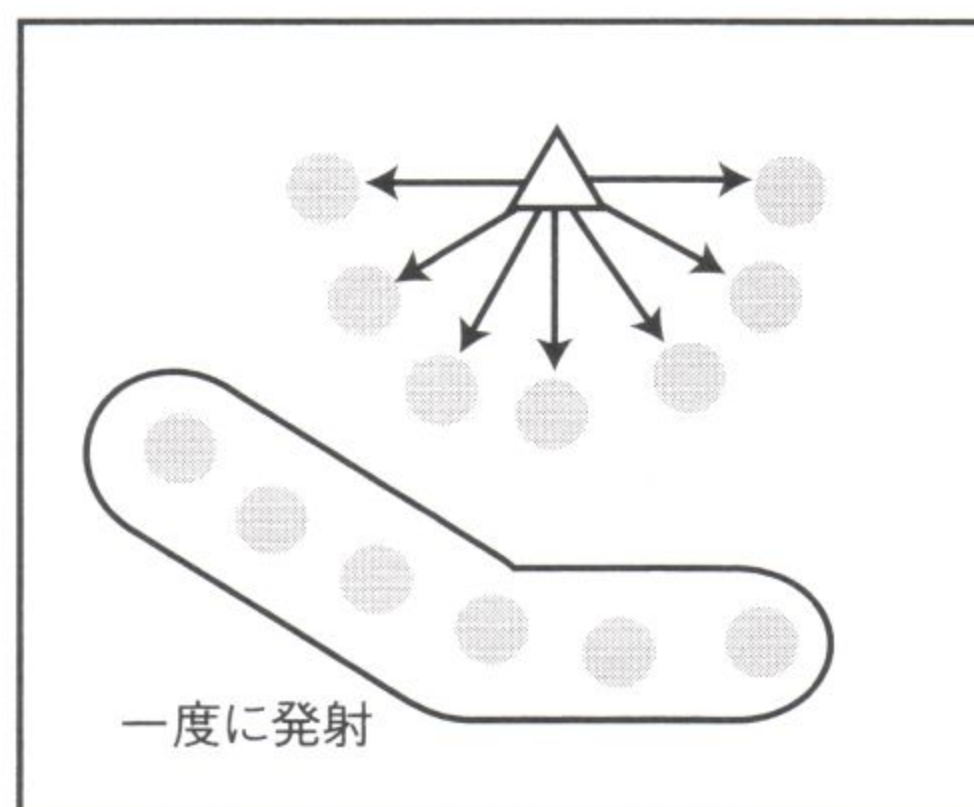
7-9 敵の弾を大量に発射する

大量の弾を撃つ

弾を大量に発射する処理を作成してみましょう。

前項のサンプルでは、単発の弾を処理する事を前程としていたため、1フレームに1発の弾しか発射できませんでした。

ここでは、そのポイントを改良し1フレームの間に複数の弾を発射するようにしてみます。

図7－9－1 1フレームに複数の弾を打って大量の弾を出すイメージ図

一度に1発ずつ撃つのでは
連続して撃っても限界がある

一度に多数の弾を発射し
大量に弾を打てるようにする

16方向へ弾を撃つプログラム

では、実際のプログラムを見ていきます。

サンプルは、一定時間毎に全方位、16方向へ弾を発射するものです。

初期化後、メインの処理は、一定時間毎に複数の弾を生成します。弾はタスクで構成され、1発の弾＝1つのタスクです。

16方向へ出す方法

この時、弾1発毎に、方向を少しづつずらした値を渡します。

ずらす値は、全方位の角度を弾の発射数で割った物で、こうする事で自動的に全方位に向けて発射されます。

弾の処理

次に、弾の処理です。まず弾を発射された瞬間に、座標、及び移動の増分値の初期化処理を行ないます。

座標は画面中央、増分値はメインの処理から渡された方向値を元に、sin、cos 関数を用いて求めます。

初期化処理が終了したら、あとは、毎フレーム増分値を座標に加算してやるだけです。

最後に一定時間経過したら弾を消去し、処理は終了です。

LIST 7 －9－1 弾を大量に発射する

```
typedef struct{
    SPRITE          sprt;
    int             Time;
    float           Direction;          //弾の方向
    float           AddX;               //増分値X
    float           AddY;               //増分値Y
} EX07_10_STRUCT ;

void exec07_09_bullet(TCB* thisTCB)
{
#define MOVE_SPEED  16.0
    EX07_09_STRUCT* work = (EX07_09_STRUCT*)thisTCB->Work;

    if( work->Time == 0 )
    {   //座標の初期化
        work->sprt.X = SCREEN_WIDTH  / 2;
        work->sprt.Y = SCREEN_HEIGHT / 2;

        //方向から、X、Yそれぞれの座標増分値を計算
        work->AddX = cos( work->Direction ) * MOVE_SPEED;
        work->AddY = sin( work->Direction ) * MOVE_SPEED;
    }
    //一定時間たったら消去する
    if( work->Time == 30 )
    {//弾を消去
        TaskKill( thisTCB );
    } else {
     //弾の進行処理
```

```
        work->Time++;
        //座標に増分値を加算
        work->sprt.X += work->AddX;
        work->sprt.Y += work->AddY;

        SpriteDraw(&work->sprt,1);
    }
}

void exec07_09(TCB* thisTCB)
{
#define BULLET_CYCLE 15
#define BULLET_COUNT 16
    int* work_time = (int*)thisTCB->Work;
    EX07_09_STRUCT* tmp_work;
    TCB*    tmp_tcb;
    int     loop;
    float   bullet_direction = 0;

    //一定時間ごとに弾を発射
    if( *work_time == BULLET_CYCLE )
    {
        //弾を発射
        for( loop = 0;loop < BULLET_COUNT; loop++ )
        {
            tmp_tcb = TaskMake( exec07_09_bullet, 0x2000 );
            tmp_work = (EX07_09_STRUCT*)tmp_tcb->Work;
            tmp_work->Direction = bullet_direction;

            //発射方向を弾の発射数に合わせて少しづつずらしていく
            bullet_direction += M_PI * 2.0  / BULLET_COUNT;
        }
        *work_time = 0;
    }

    *work_time += 1;
}
```

7-10　自機の弾を撃つ

自機が弾を撃つプログラムの概要

自機から弾を発射する処理を作成してみましょう。
シンプルな処理ですが、シューティングをはじめとして色々な処理の基本でもあります。
では早速プログラムを見ていきましょう。

まず、フレームをまたがって保存しておく変数の定義です。
タスクに割り当てて使用する構造体には、自機と発射する弾を管理するスプライトを定義します。
同様に弾の発射モードを切り替えるフラグを用意しておきます。今回使用する変数は以上です。

初期化

次に初期化を行ないます。使用するグラフィックデータを読み込んだ後、自機の座標を初期化します。
座標は画面中央にしているため、画面の幅と高さを２で割ったものを代入します。

メイン処理

次は、メインの処理です。関数のはじめに、自機の移動速度と、弾の速度をマクロで定義しています。
処理の方は開始後、すぐに自機の移動を行なっています。これは、キー入力を見て、各方向に自機の移動速度を加算するだけです。
ただ、この手法は手軽なのですが、方向によって移動速度に違和感が出ます。
そのため厳密な処理をする場合は、[7－2]を参考にして下さい。

弾を撃つ処理

そして弾の発射処理です。ここでは先ほど用意した弾の発射モードを切り替えるフラグ、BulletDrawを参照しています。
ボタンが押された時に発射可能であれば、このフラグを立てて、弾の移動、表示をするモードに切り替えます。

このように同じタスク処理関数内で、フラグや状態を見て処理モードを切り替えるのは頻繁に使われるテクニックです※。
なお、弾は自機の座標から発射されるため、自機の座標を弾の座標にコピーする事も忘れないで下さい。

※状態自体が大幅に変わる場合は、モードの切り替えでは対処しづらいため、TaskChangeを使用します。

弾の移動と表示

次に、弾の移動処理と表示です。弾は画面上方に飛ぶため、移動は弾の座標から、BULLET_SPEEDを減算する事で行ないます。

処理後、画面外に弾が移動していれば、弾の処理が終了したとみなし、モードを切り替えてキー入力待ち状態に戻ります。

最後は自機の表示をして、処理の終了となります。

LIST 7 －10－1 弾を撃つ

```
typedef struct{
    SPRITE          Bullet;
    SPRITE          MyShip;
    int             BulletDraw;
} EX07_10_STRUCT ;

void init07_10(TCB* thisTCB)
{
    EX07_10_STRUCT* work = (EX07_10_STRUCT*)thisTCB->Work;

    //使用するテクスチャの読み込み
    D3DXCreateTextureFromFile( g_pD3Ddevice, "..¥¥..¥¥data¥¥0044.png",&g_pTex[0] );
    D3DXCreateTextureFromFile( g_pD3Ddevice, "..¥¥..¥¥data¥¥0055.png",&g_pTex[1] );

    work->MyShip.X = SCREEN_WIDTH  / 2;
    work->MyShip.Y = SCREEN_HEIGHT / 2;

}

void exec07_10(TCB* thisTCB)
{
#define MOVE_SPEED    4.0
#define BULLET_SPEED  10.0
    EX07_10_STRUCT* work = (EX07_10_STRUCT*)thisTCB->Work;

    //キー入力による自機の移動
    if( g_InputBuff & KEY_UP    ) work->MyShip.Y -= MOVE_SPEED;
    if( g_InputBuff & KEY_DOWN  ) work->MyShip.Y += MOVE_SPEED;
    if( g_InputBuff & KEY_RIGHT ) work->MyShip.X += MOVE_SPEED;
    if( g_InputBuff & KEY_LEFT  ) work->MyShip.X -= MOVE_SPEED;
```

```
    if( !work->BulletDraw )
    {//発射キー入力待ち
      if( g_DownInputBuff & KEY_Z )
      {//弾の発射処理
          //自機の座標をコピーして、そこから弾を発射する
          work->Bullet.X = work->MyShip.X + 16;
          work->Bullet.Y = work->MyShip.Y;

          work->BulletDraw = true;
      }
    }else{
       //弾の移動処理と表示
        work->Bullet.Y -= BULLET_SPEED;
        SpriteDraw( &work->Bullet,1);

        //弾が画面外に移動したら、キー入力待ちに戻る
        if( work->Bullet.Y < 0 ) work->BulletDraw = false;
    }

    SpriteDraw( &work->MyShip, 0);
}
```

7-11 複数の弾を連続して発射する

複数の弾の管理

[7－10]では単発の弾しか発射出来ませんでした。インベーダーのようなシンプルなゲームならば、問題は無いのですが、複数の弾を連発するゲームだと、あの手法では少々無理があります。

そこでここでは、並列処理システムを利用し、タスクによる複数の弾の発射を行なってみます。

すなわち、ボタンが押された時に弾の発射モードに移行するのではなく、タスクを作成する事で処理を行ないます。

複数の弾を発射する処理

ではプログラムを見ていきましょう。

まず、初期化処理の後、メインの処理に移ります。この時、自動変数にタスクを作成した時に処理を行なうための変数を用意しておきます。

処理の方は、最初に自機の移動を行ないます。その後ボタンチェックを行ない、弾の発射処理を行ないます。

タスクは関数TaskMakeを用いて行なわれます。この関数は作成したタスクへのポインタを返しますので、先ほど用意した変数に格納しておきます。

作成後、このポインタを用いて、自機の座標を弾のタスクにコピーします。

メイン側の処理は以上で終了です。

次に弾の処理関数exec07_11_bulletを見ていきます。

まず弾が、画面外に位置しているかどうかチェックします。もし画面外に位置していたら処理は終了します。

もし、そうでなければ、弾の進行処理と表示を行ないます。以上で弾の処理は終了です。

最後に、サンプルではプログラム自体を、[7－10]の処理に似せるようにしています。理解が難しいようでしたら、是非何処が違うかを見比べてみてください。

LIST 7 －11－1 複数の弾を連続して発射する

```
typedef struct{
    SPRITE          MyShip;
} EX07_11_STRUCT ;

#define MOVE_SPEED    4.0
#define BULLET_SPEED  10.0

void exec07_11_bullet(TCB* thisTCB)
{
    SPRITE* work = (SPRITE*)thisTCB->Work;

     //弾が画面外に移動したら、処理の終了
    if( work->Y < 0 )
    {
        TaskKill(thisTCB);
        return;
    }

    //弾の移動処理と表示
    work->Y -= BULLET_SPEED;
    SpriteDraw( work,1);
}

void init07_11(TCB* thisTCB)
{
    EX07_11_STRUCT* work = (EX07_11_STRUCT*)thisTCB->Work;

    //使用するテクスチャの読み込み
    D3DXCreateTextureFromFile( g_pD3Ddevice, "..¥¥..¥¥data¥¥0044.png",&g_pTex[0] );
    D3DXCreateTextureFromFile( g_pD3Ddevice, "..¥¥..¥¥data¥¥0055.png",&g_pTex[1] );

    work->MyShip.X = SCREEN_WIDTH  / 2;
    work->MyShip.Y = SCREEN_HEIGHT / 2;
}

void exec07_11(TCB* thisTCB)
{
```

```
    EX07_11_STRUCT* work = (EX07_11_STRUCT*)thisTCB->Work;
    TCB*     bullet_tcb;
    SPRITE* bullet_work;

    //キー入力による自機の移動
    if( g_InputBuff & KEY_UP    ) work->MyShip.Y -= MOVE_SPEED;
    if( g_InputBuff & KEY_DOWN  ) work->MyShip.Y += MOVE_SPEED;
    if( g_InputBuff & KEY_RIGHT ) work->MyShip.X += MOVE_SPEED;
    if( g_InputBuff & KEY_LEFT  ) work->MyShip.X -= MOVE_SPEED;

    if( g_DownInputBuff & KEY_Z )
    {//弾の発射処理
        //タスクを使用して弾を作成する
        bullet_tcb = TaskMake( exec07_11_bullet, 0x2000 );
        bullet_work = (SPRITE*)bullet_tcb->Work;

        //自機の座標をコピーして、そこから弾を移動させる
        bullet_work->X = work->MyShip.X + 16;
        bullet_work->Y = work->MyShip.Y;
    }

    SpriteDraw( &work->MyShip, 0);
}
```

7-12 発射に対して反動をつける

反動をつける演出

ここでは、弾の発射に対して反動を付ける事を考えてみます。

反動の処理を大きな弾や、タメ撃ちなどをした時などに加えると、ずっとリアルに見えます。

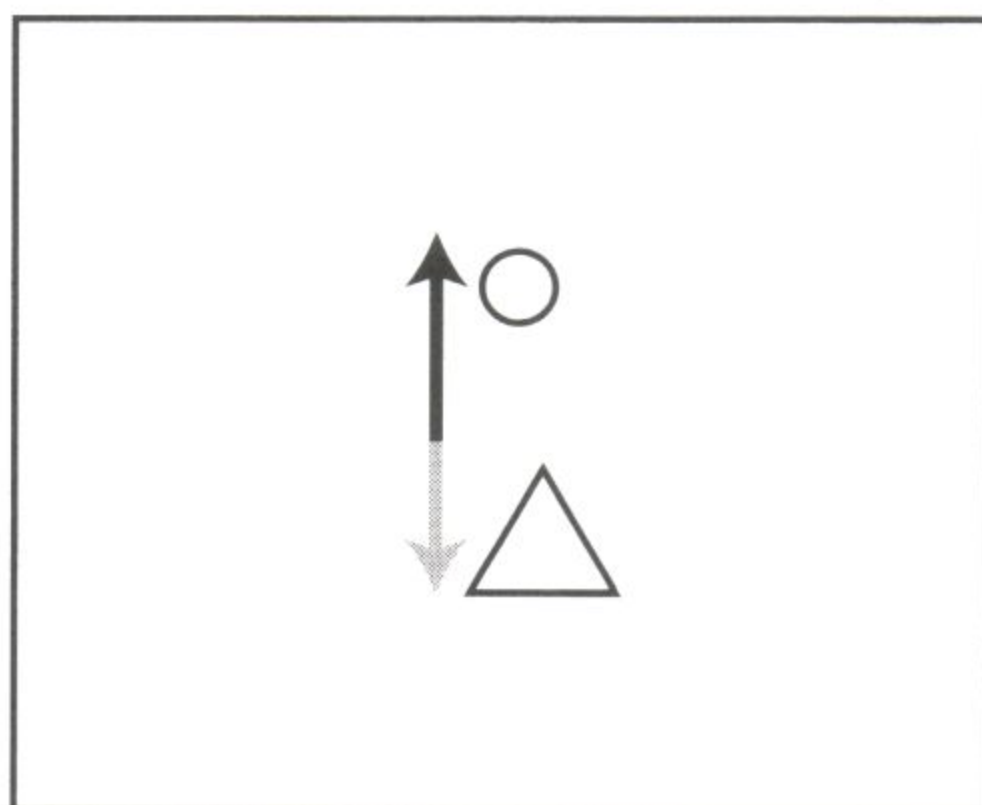
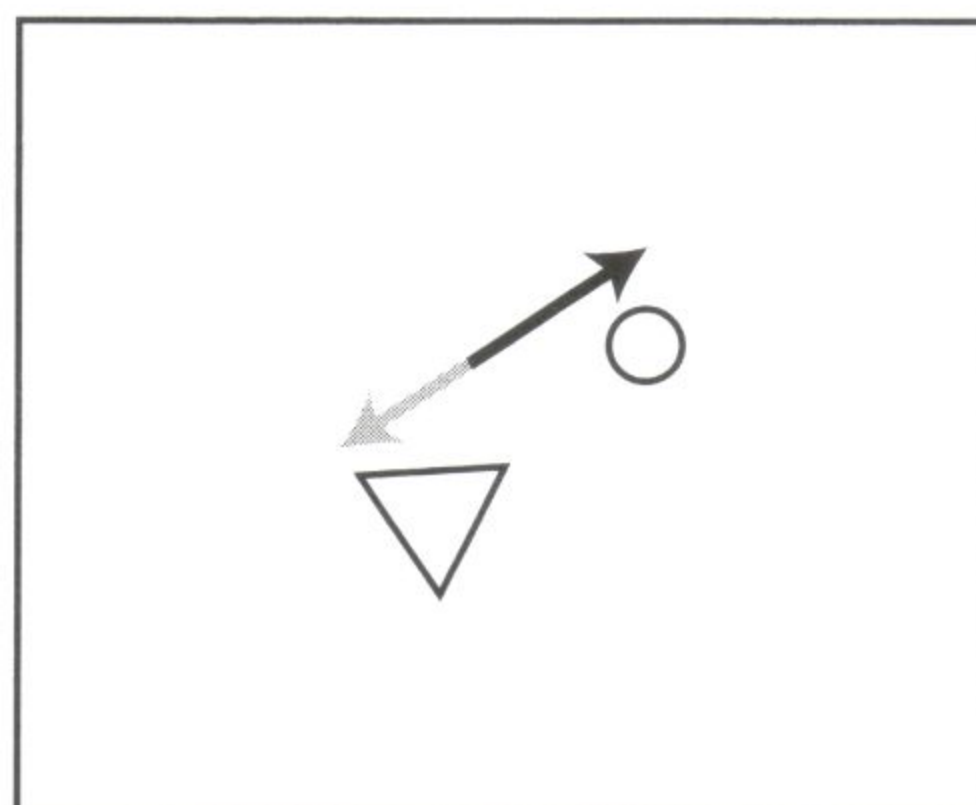
また、操作性にも若干影響を与えるため、うまく作れば、面白い操作感覚が得られます。

その反動の処理ですが、弾を発射する時に、発射方向の反対へ自機移動させる事で行ないます。

この時、どの程度移動させるか？　また、移動中に操作が可能かどうか？　で操作間隔が変わってきます。

また、反対方向の値は進行方向のX、Yの値を反転させる事で得られます。

これは、方向がX、Y軸に対して垂直な時だけではなく、全ての発射方向に対して有効です。

図7 －12－1 進行方向に対して反転をさせると反対方向を向くイメージ図

発射の進行方向を反転させると
反対方向を向く

これは、どの方向に対しても同じ

反動をつけるプログラム

では実際のプログラムを見ていきましょう。

初期化後、まず自機の移動を行ないますが、この時に変数、LockTimeを見るようにします。

これは弾の発射時、反動で移動している時間を表す変数で、この値が0以外であれば、操作を受けつけず、発射方向の反対側へ移動するようにします。

次に弾の発射処理です、ここの処理自体は、[7－11]とほとんど変わりません。ボタンが押され

た瞬間に弾の処理タスクを作成し、自機の座標をコピーしています。

そして、この時に同時に操作が不可能な時(=反動を行なう時間)を設定しています。

後は自機の表示を行ない、処理は終了です。

なお、弾の移動処理自体は、[7－11]と同じになっていますので、ここでの解説は避けます。

LIST 7－12－1 発射に対して反動を付ける

```
typedef struct{
    SPRITE         MyShip;
    int            LockTime;
} EX07_12_STRUCT ;

#define MOVE_SPEED    4.0
#define BULLET_SPEED  10.0
#define LOCK_TIME     4

void exec07_12_bullet(TCB* thisTCB)
{
    SPRITE* work = (SPRITE*)thisTCB->Work;

     //弾が画面外に移動したら、処理の終了
    if( work->Y < 0 )
    {
        TaskKill(thisTCB);
        return;
    }

    //弾の移動処理と表示
    work->X += 0;
    work->Y += -BULLET_SPEED;
    SpriteDraw( work,1);
}

void init07_12(TCB* thisTCB)
{
    EX07_12_STRUCT* work = (EX07_12_STRUCT*)thisTCB->Work;

    //使用するテクスチャの読み込み
   D3DXCreateTextureFromFile( g_pD3Ddevice, "..¥¥..¥¥data¥¥0044.png",&g_pTex[0] );
   D3DXCreateTextureFromFile( g_pD3Ddevice, "..¥¥..¥¥data¥¥0055.png",&g_pTex[1] );
```

```
    work->MyShip.X = SCREEN_WIDTH  / 2;
    work->MyShip.Y = SCREEN_HEIGHT / 2;
}

void exec07_12(TCB* thisTCB)
{
    EX07_12_STRUCT* work = (EX07_12_STRUCT*)thisTCB->Work;
    TCB*    bullet_tcb;
    SPRITE* bullet_work;

    if( !work->LockTime )
    {//操作可能
        //キー入力による自機の移動
        if( g_InputBuff & KEY_UP    ) work->MyShip.Y -= MOVE_SPEED;
        if( g_InputBuff & KEY_DOWN  ) work->MyShip.Y += MOVE_SPEED;
        if( g_InputBuff & KEY_RIGHT ) work->MyShip.X += MOVE_SPEED;
        if( g_InputBuff & KEY_LEFT  ) work->MyShip.X -= MOVE_SPEED;
    }else{
     //操作不可能時間
        work->LockTime--;
        //弾の発射方向とは反対方向に進む
        work->MyShip.X += -0.0;
        work->MyShip.Y += BULLET_SPEED / 4;
    }

    if( g_DownInputBuff & KEY_Z )
    {//弾の発射処理
        //タスクを使用して弾を作成する
        bullet_tcb = TaskMake( exec07_12_bullet, 0x2000 );
        bullet_work = (SPRITE*)bullet_tcb->Work;

        //自機の座標をコピーして、そこから弾を移動させる
        bullet_work->X = work->MyShip.X + 16;
        bullet_work->Y = work->MyShip.Y;
        //操作不可能時間を設定
        work->LockTime = LOCK_TIME;
    }

    SpriteDraw( &work->MyShip, 0);
}
```

7-13 レーザーの動き

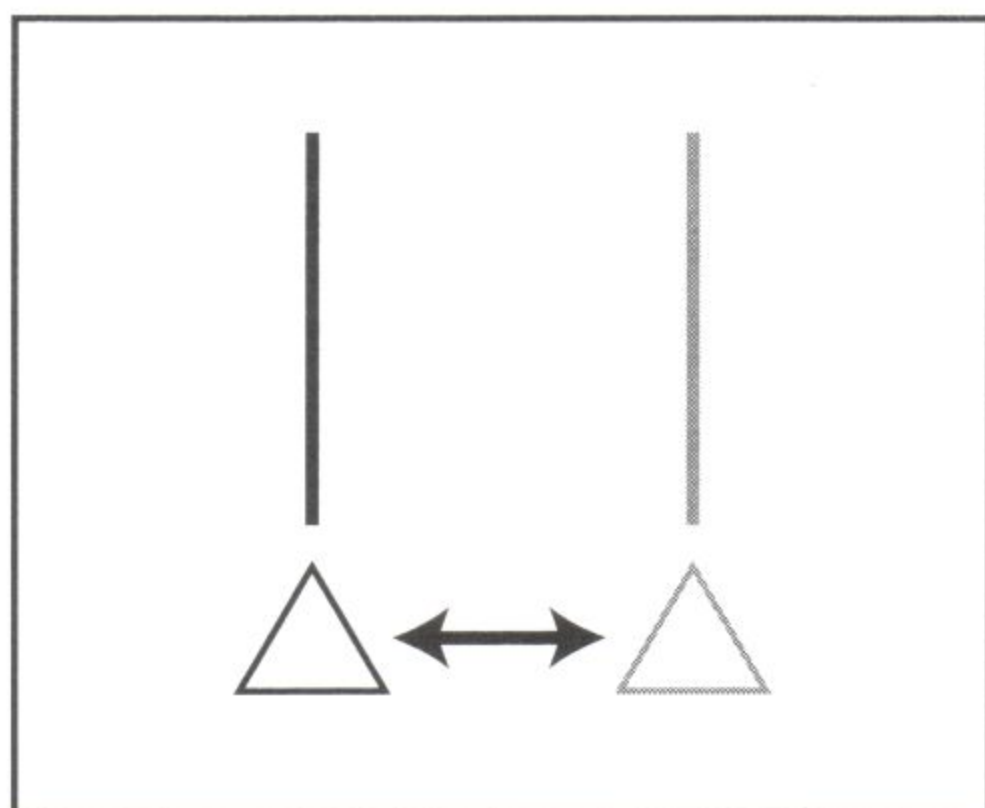
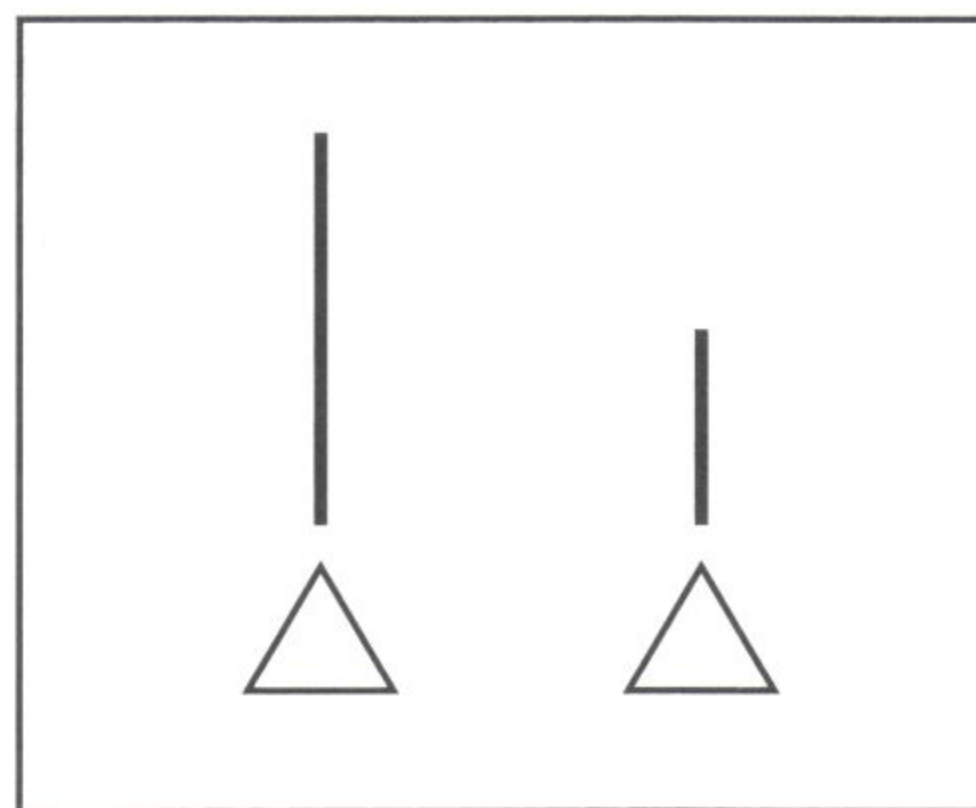
シューティングゲームのレーザー

シューティングゲームでよく見られるレーザーの動きを再現してみましょう。

自機から発射されるレーザーの動きにはゲームによって幾つもパターンがあり、全てを再現するのは少し難しい所ですが、ここでは代表的な動きとして、以下の２つの処理を実装してみます。

・自機の動きに合わせて移動（ワインダー処理）

・ボタンを押す時間に応じて、レーザーの長さを変える

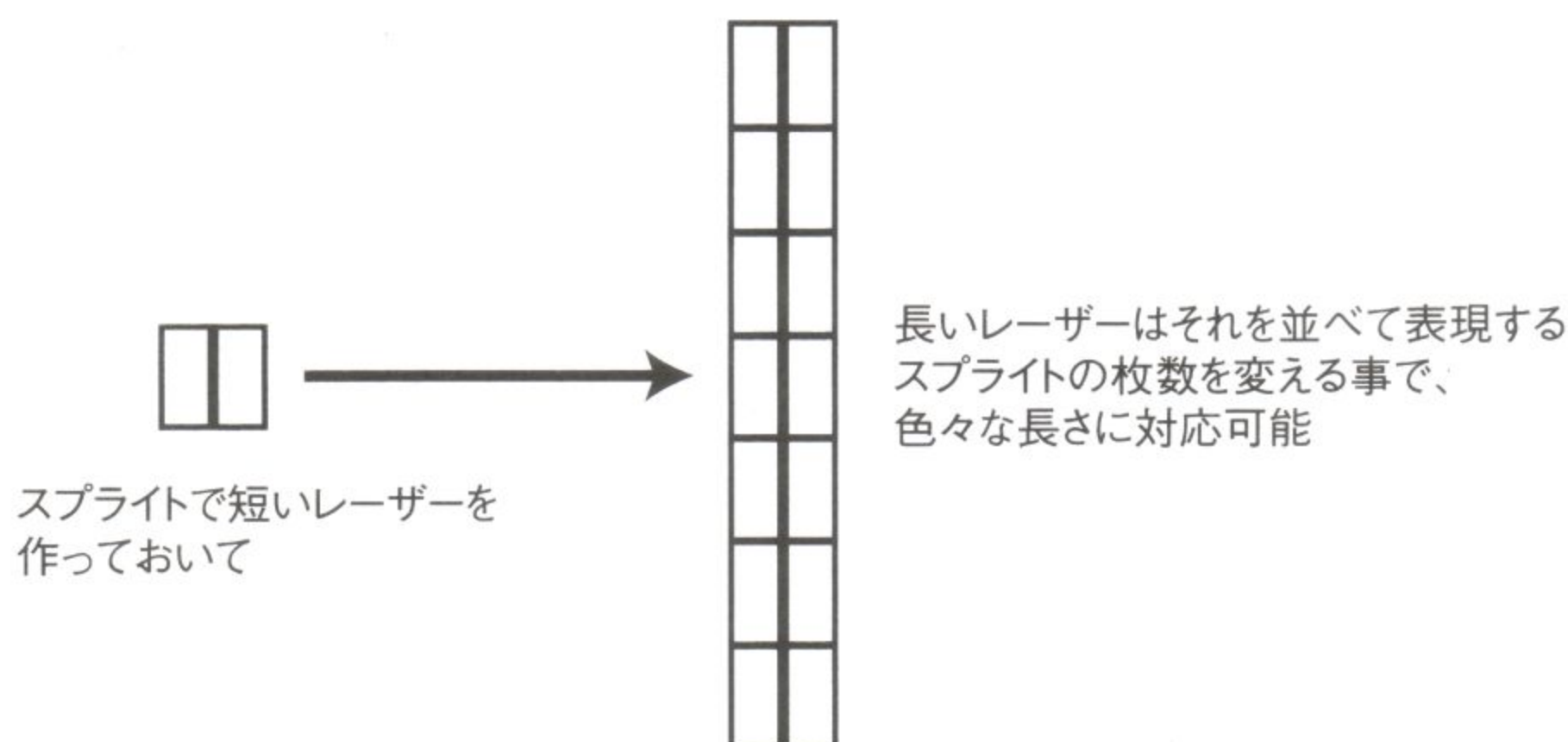
図7－13－1 レーザー処理のイメージ図

自機の移動に会わせて、レーザーも移動するワインダー処理

押す時間に会わせてレーザーの長さが変わる

レーザーの表現処理

このレーザーの処理概要ですが、スプライトで行なう方法と、線の描画命令を使用して行なう方法があります。

どちらにも一長一短がありますが、ここではグラフィックを変更するだけで、色々な形状のレーザーを定義できる前者の方法で表示を行ないます。

処理としては、短いレーザーの形状をスプライトで定義し、押した長さによって表示するスプライトの枚数を変更します。

図7 －13－2　スプライトの枚数を変更して、レーザーの長さを変えるイメージ図

この時、画面上には１つのレーザーのみを出すため、レーザーの表示状態を管理するフラグを設けます。

そうしないと、ボタンを連打した際、細切れにレーザーが出てしまうからです。

レーザーを撃つプログラム

それでは実際のプログラムを見ていきましょう。

サンプルの動作は自機を動かし、レーザーを発射する単純な物です。

初期化処理後、メインの処理では自機の移動と、レーザーの発射管理を行ないます。

レーザーの発射管理は、多数のワークで行なうため少々ややこしいので、順を追って説明します。

ボタンを押したときのチェック

まず、レーザーの発射のためのボタン押下のチェックを行ないます。

この際、もし画面上に前回発射されたレーザーが残っていたら何もしません。ここでチェックをする事により、画面上に表示されるレーザーは常に１つのみになります。

画面上にレーザーが無い場合は、即座に発射するのではなく、レーザーの発射モードに移行します。この時レーザーの長さをカウントするワーク等も初期化します。

レーザー発射処理

次に、実際のレーザーの発射です。

レーザーモードの時に、もしレーザーが発射可能な状態であれば、常にレーザーを発射し続けます。

レーザーが発射可能な状態とは、レーザーの長さが一定値を越えていない事（マクロLASER_LENGTHで定義）、レーザー発射時からボタンが離されていない事、の２つです。

もしこの条件が崩れるようであれば、レーザー発射のモードはOFFとなり、画面上からレーザーがなくなるまで、レーザーが発射される事はありません。

実際のレーザーの発射ですが、タスクを使用して行なわれ、上記の条件が満たされている間、ずっと作成し続けます。

すなわち レーザーの長さ＝レーザー処理タスクの数 です。

この時作成と同時に、レーザーが画面に残っている事を示す表示フラグをオンにします。

レーザーの動き

最後にレーザーそのものの処理です。

レーザーは自機の動きを追従する動き（ワインダー処理）を行なうため、自機の座標へのポインタを保持しています。

この時、レーザーの座標は自機との相対距離となるため、表示座標とは別に移動のための相対座標を計算、保持しておきます。

自機の座標に、この相対座標を加算してやれば、自機に追従した移動＝ワインダー処理となります。

レーザーの消去

次にレーザーの消去処理です。レーザー自体は画面上方に進むだけですので、Y座標が一定値以下になったかどうかで、画面外を判定する事が出来ます。

もし画面外であれば、レーザーの処理タスクを終了します。

この時、画面上のレーザーの表示フラグを管理するため、レーザーを幾つ消したかをカウントしておき、作成したレーザーの数と比較します。

もし同じであれば画面上からレーザーは消えていますので、レーザーの表示フラグはオフになります。この処理でようやくレーザーの再発射が可能となります。

LIST 7 －13－1 レーザーの動き

```
#define LASER_LENGTH 10
#define LASER_SPEED  16
typedef struct{
    SPRITE            MyShip;              //自機
    int               LaserCount;          //発射したレーザーの数
    int               DeleteLaserCount;    //消去したレーザーの数
    int               LaserMode;           //レーザーの発射モード
    int               LaserDraw;           //レーザーが画面内に残っているかのフラグ
} EX07_13_STRUCT;

typedef struct{
    EX07_13_STRUCT*   MyShipWork;      //自機情報へのポインタ
    float             PosY;            //自機からの相対座標
    SPRITE            Laser;           //レーザー表示用スプライト
```

```
} EX07_13_LASER;

void exec07_13_laser(TCB* thisTCB)
{
    EX07_13_LASER* work = (EX07_13_LASER*)thisTCB->Work;

    //レーザーを進める
    work->PosY -= LASER_SPEED;
    //レーザーの座標を自機の座標に追従させる
    work->Laser.X = work->MyShipWork->MyShip.X + 24;
    work->Laser.Y = work->PosY + work->MyShipWork->MyShip.Y;

    if( work->Laser.Y > -16 )
    {//レーザーの描画
        SpriteDraw( &work->Laser, 1);
    }else{
    //レーザーが画面外なら表示されている消去されたレーザーの数をカウントし消去
    //もしその際、作成したレーザーの数と消去したレーザーの数が同じなら画面からレーザーは消えている。
        work->MyShipWork->DeleteLaserCount++;
        if( work->MyShipWork->LaserCount == work->MyShipWork->DeleteLaserCount)
            work->MyShipWork->LaserDraw = FALSE;
        TaskKill( thisTCB );
    }
}

void init07_13(TCB* thisTCB)
{
    //使用するテクスチャの読み込み
    D3DXCreateTextureFromFile( g_pD3Ddevice, "..¥¥..¥¥data¥¥0044.png",&g_pTex[0] );
    D3DXCreateTextureFromFile( g_pD3Ddevice, "..¥¥..¥¥data¥¥LASER.png",&g_pTex[1] );
}

void exec07_13(TCB* thisTCB)
{
#define MOVE_SPEED  4.0
    EX07_13_STRUCT* work = (EX07_13_STRUCT*)thisTCB->Work;
    TCB*  tmp_tcb;
    EX07_13_LASER*  tmp_work;
```

```
    //キー入力による自機の移動
    if( g_InputBuff & KEY_UP    ) work->MyShip.Y -= MOVE_SPEED;
    if( g_InputBuff & KEY_DOWN  ) work->MyShip.Y += MOVE_SPEED;
    if( g_InputBuff & KEY_RIGHT ) work->MyShip.X += MOVE_SPEED;
    if( g_InputBuff & KEY_LEFT  ) work->MyShip.X -= MOVE_SPEED;

    //発射キーが押されたらレーザー発射モードに切り替える
    if( g_DownInputBuff & KEY_Z )
    {//但し、すでに発射されたレーザーが画面内に残っていたら何もしない。
      if( !work->LaserDraw )
      {//レーザー発射関連の初期化
        work->LaserMode = TRUE;
        work->LaserCount = 0;
        work->DeleteLaserCount = 0;
      }
    }

    //レーザー発射モードであればレーザーを発射し続ける
    if( work->LaserMode )
    {//但し、レーザーの長さが一定量を超えているか、ボタンが離された場合、レーザーモードは終了
        if( work->LaserCount > LASER_LENGTH || g_UpInputBuff & KEY_Z )
        {
            work->LaserMode = FALSE;
        } else {
            //レーザー表示処理の作成
            work->LaserCount++;
            tmp_tcb = TaskMake( exec07_13_laser, 0x2000 );
            tmp_work = (EX07_13_LASER*)tmp_tcb->Work;
            tmp_work->MyShipWork = work;
            //レーザー表示中
            work->LaserDraw = TRUE;
        }
    }

    SpriteDraw( &work->MyShip, 0);
}
```

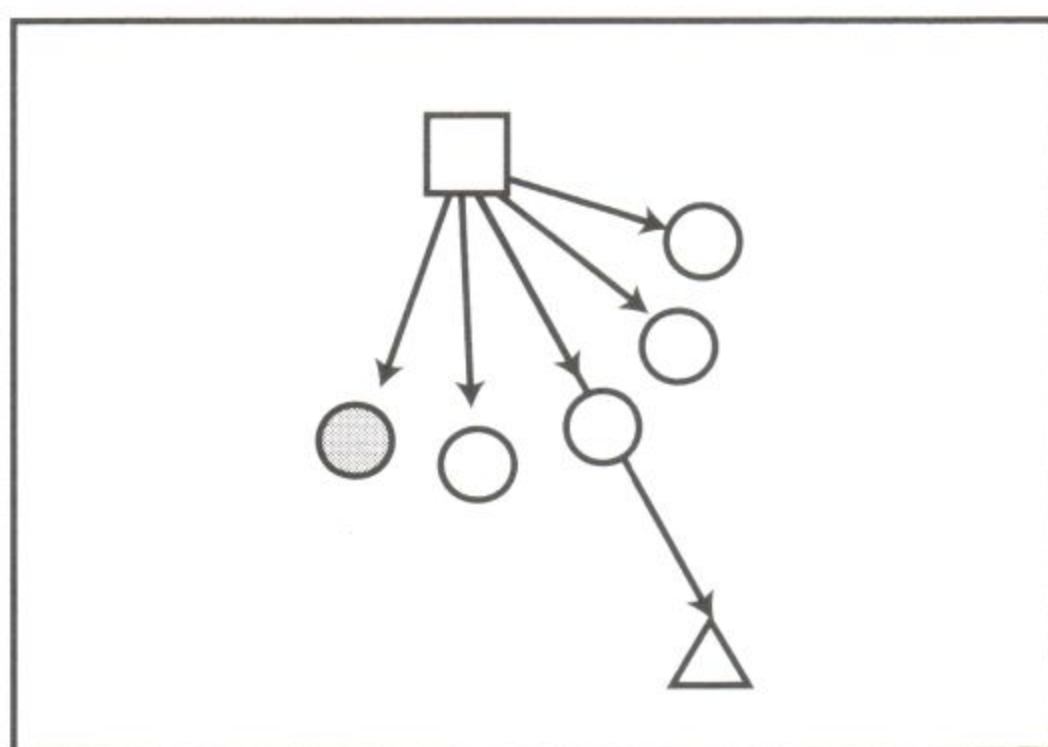
7-14 多方向へ弾を発射する

nウェイ弾

狙った方向へ弾を多数発射する処理を作成してみましょう。

これはシューティング等で、扇状に弾を発射する「nウェイ(n-way)弾」として知られています。

また、自機の攻撃として使用する以外にも、相手の攻撃や扇状に広がるエフェクトとして色々利用できます。

図7 －14－1 nウェイ弾のイメージ図

相手を狙いつつ、扇状に弾を発射する

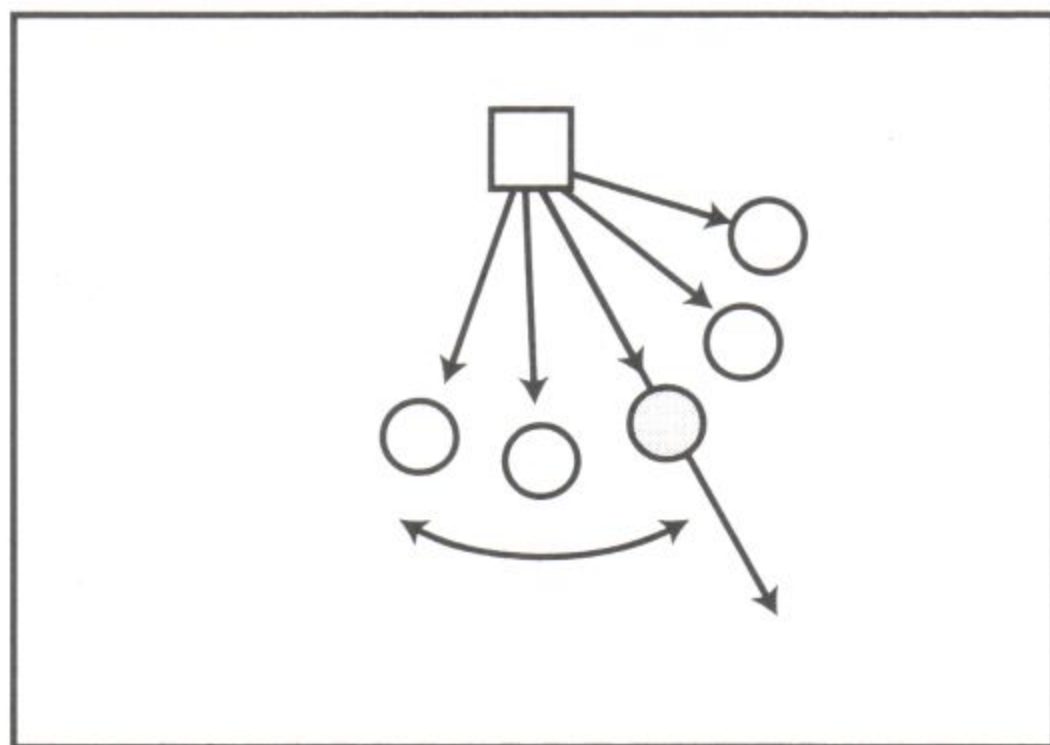
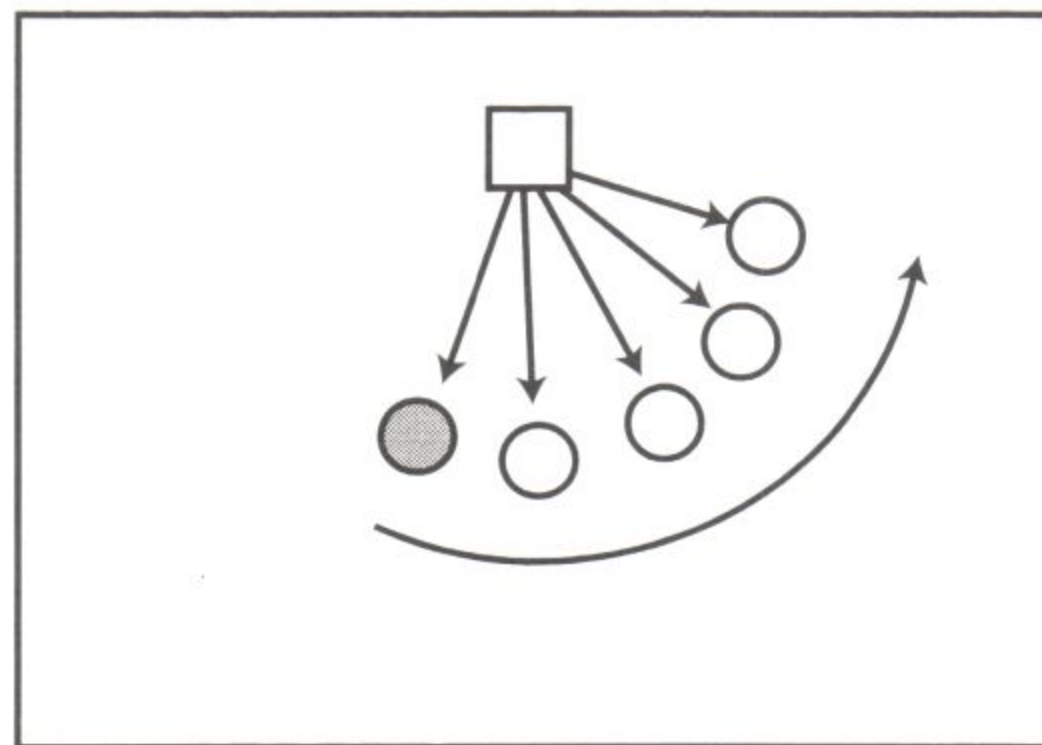
nウェイ弾の処理

処理の概要から解説していきます。まず、相手を狙う弾を考えてください。

そして、その弾に対して一定量の角度をずらした弾を発射するようにします。この角度が扇の幅になります。

ただし、そのままだとずらす処理を、弾の左右両方向に対して行なわなくてはならないため、処理が若干面倒です。

そこで、はじめに一気に角度をずらしてやり、そこから、一方向に対してずらすようにしてやります。

図7－14－2 左右にずらすのではなく、片一方に一気にずらして処理を簡略化するイメージ図

そのままだと、中心となる弾を基準に
作成処理を左右に広げないといけない

作成開始の弾を片一方にずらし
一気に作成処理を行なう

nウェイ弾のプログラム

5方向に発射

では実際のプログラムを解説します。サンプルは画面中央から5方向の扇状弾が、一定時間毎に発射されるものです。

プログラムではデータの初期化後、自機移動の処理と、弾発射を管理する処理を作成しています。

弾発射の処理は、自機の座標を取得する必要があるため、自機の作業ワークのポインタを渡しています。

初期化

実際の処理ですが、まず一定時間毎に弾を発射するために、タイマーをチェックしています。

チェック後、弾が発射されるようなら、弾の初期化処理に移ります。

最初に目標の座標と弾の発射座標を取得します。サンプルでは発射座標は中央固定ですので、画面の幅と高さを2分した値を使用します。

発射方向

次に得られた座標から弾を発射する方向を取得し、得られた方向に対して発射の方向を、弾の発射数÷2個分ずらしてやります。

こうする事で弾作成のループが単純になり、処理が簡略化されます。

弾の作成

最後に弾の作成ですが、ここは発射数分のループとなっています。

弾を作成後、先ほど計算した発射方向から弾の移動値を計算しています。

この時、弾を1発作成するごとに発射方向を、少しずつずらしていきます。

以上で処理は完成です。

LIST 7 −14−1 多方向へ弾を発射する

```
typedef struct{
    SPRITE*         Target;             //目標のスプライト
    int             Time;
} EX07_14_STRUCT;

typedef struct{
    SPRITE          Sprt;
    int             Time;
    float           AddX;               //増分値X
    float           AddY;               //増分値Y
} EX07_14_BULLET;

void exec07_14_bullet(TCB* thisTCB)
{
    EX07_14_BULLET* work = (EX07_14_BULLET*)thisTCB->Work;

    //一定時間たったら消去する
    if( work->Time == 30 )
    {//弾を消去
        TaskKill( thisTCB );
    } else {
     //弾の進行処理

        work->Time++;
        //座標に増分値を加算
        work->Sprt.X += work->AddX;
        work->Sprt.Y += work->AddY;

        SpriteDraw(&work->Sprt,1);
    }
}

void exec07_14_shoot(TCB* thisTCB)
{
#define BULLET_CYCLE 15
#define BULLET_COUNT 5
#define BULLET_ANGLE M_PI / 16      //弾と弾の間の角度
#define MOVE_SPEED  16.0

```

```
    EX07_14_STRUCT* work = (EX07_14_STRUCT*)thisTCB->Work;
    EX07_14_BULLET* tmp_work;
    TCB*    tmp_tcb;
    int     loop;
    //弾の発射座標
    float shootX;
    float shootY;
    //目標の座標
    float targetX;
    float targetY;
    //弾を発射する方向
    float bullet_direction;

    //一定時間ごとに弾を発射
    if( work->Time == BULLET_CYCLE )
    {
        //画面中央から弾を発射
        shootX = SCREEN_WIDTH  / 2;
        shootY = SCREEN_HEIGHT / 2;
        //目標の座標を取得
        targetX = work->Target->X + 16;
        targetY = work->Target->Y + 16;

        //座標から弾の発射方向を計算
        bullet_direction = atan2( targetY - shootY, targetX - shootX );
        //最初に一気に方向を引いておくこの場合は5WAYなので2発分の角度を引く
        bullet_direction -= BULLET_ANGLE * (BULLET_COUNT / 2) ;
        //弾を発射
        for( loop = 0;loop < BULLET_COUNT; loop++ )
        {
            tmp_tcb = TaskMake( exec07_14_bullet, 0x2000 );
            tmp_work = (EX07_14_BULLET*)tmp_tcb->Work;

            //方向から、X、Yそれぞれの座標増分値を計算
            tmp_work->AddX = cos( bullet_direction ) * MOVE_SPEED;
            tmp_work->AddY = sin( bullet_direction ) * MOVE_SPEED;
            //発射方向を少しづつずらしていく
            bullet_direction += BULLET_ANGLE;
            //弾の初期座標
            tmp_work->Sprt.X = shootX;
```

```
            tmp_work->Sprt.Y = shootY;
        }
        work->Time = 0;
    }
    work->Time += 1;
}

void init07_14(TCB* thisTCB)
{
    TCB*    tmp_tcb;
    SPRITE* sprt;
    EX07_14_STRUCT* tmp_work;
    int loop;

    //使用するテクスチャの読み込み
    D3DXCreateTextureFromFile( g_pD3Ddevice, "..¥¥..¥¥data¥¥0044.png",&g_pTex[0] );
    D3DXCreateTextureFromFile( g_pD3Ddevice, "..¥¥..¥¥data¥¥0055.png",&g_pTex[1] );

    //自機のスプライト
    sprt = (SPRITE*)thisTCB->Work;
    sprt->X = SCREEN_WIDTH  / 2;
    sprt->Y = SCREEN_HEIGHT / 2 + 120;

    //弾発射を管理する処理を作成
    tmp_tcb = TaskMake( exec07_14_shoot, 0x2000 );
    tmp_work = (EX07_14_STRUCT*)tmp_tcb->Work;
    tmp_work->Target = sprt;

}

void exec07_14(TCB* thisTCB)
{
#define MOVE_SPEED  8.0
    SPRITE* work = (SPRITE*)thisTCB->Work;

    //キー入力による移動
    if( g_InputBuff & KEY_UP    ) work->Y -= MOVE_SPEED;
    if( g_InputBuff & KEY_DOWN  ) work->Y += MOVE_SPEED;
    if( g_InputBuff & KEY_RIGHT ) work->X += MOVE_SPEED;
    if( g_InputBuff & KEY_LEFT  ) work->X -= MOVE_SPEED;
```

```
    SpriteDraw( work, 0);
}
```

7-15 リプレイ機能

リプレイ機能の使い方

リプレイを行なう事について考えてみましょう。

リプレイ機能はゲームに置いて必須という処理ではありませんが、実装しているゲームは良く見かけます。アクション、シューティングやレースゲーム等、そのジャンルも多岐に渡ります。

リプレイの処理自体は、ゲーム性にはまったく影響しない処理ですが、何故こんなに実装されるのでしょうか。

これは、プレイヤーの見地から見ると、よく分かります。

例えば、失敗した自分のプレイを見て攻略法を考えたり、プレイを見直して気付かなかった攻略に気付くといった事はよくあります。

また、攻略以外でも上手なプレイヤーの動きを見て感心したり等、ゲームの幅を広げるためにはとても有効なのです。

そのため、ユーザーサービスという視点から見ると、是非実装したい処理といえるでしょう。

そのリプレイ処理ですが、一体どうやって実装しているのでしょうか。

リプレイ機能を実現するには

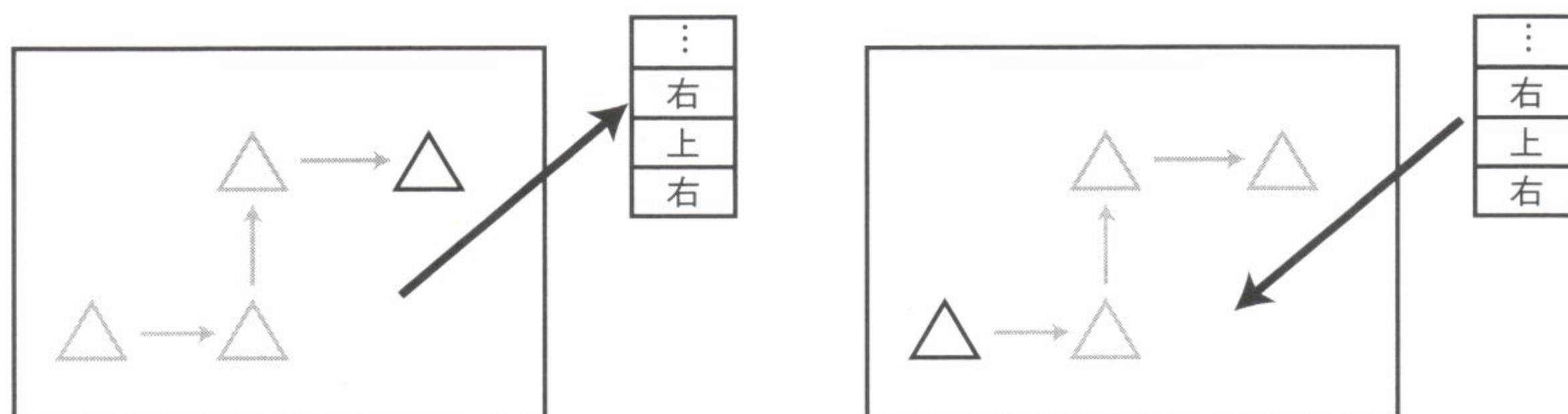
方法の１つとして、ゲームプレイ中に出てくる表示物の座標等を全て記録すると言う手法があげられます。

しかし、少し考えると分かりますが、この方法だと、とてつもない容量が必要になってしまいます。

そこでここではもう１つの手法、キー入力だけを記録しておくという手法を使います。

キー入力だけを記録しても、処理手順が同一ならば、まったく同じ様にゲームが進行します。

容量をあまり使わずに、リプレイの処理が出来るのです。

図7 －15－1 キー入力だけでリプレイが可能なイメージ図

プレイヤーの挙動をキー入力で記録しておく
記録したキー入力を

仮想のキー入力として「再生」
してやれば、同じ挙動が実現される
ただし、再現できないランダムな
要素があると失敗する

ただしこの方法を使う場合は、ランダムな要素は一切排除しないといけません。

キー入力が同じでも、ランダムな要素があると、ゲーム上での処理順序が変わってしまい、リプレイが出来なくなってしまうからです。

7-16 リプレイ機能を作る

リプレイ機能の概要

では実際に、リプレイの処理を作成していきます。

サンプルでは、Zキーを押す事で、自機の移動とキー入力の記録を、Xキーで押す事で記録した入力の再生（リプレイ）を一定時間行ないます。

なお、プレイ中は、自機を狙って画面中央から弾が発射されます。

リプレイ機能のプログラム

全体の概要

はじめにプログラム全体についてですが、大きく3つの関数に分けられます。

キー入力とリプレイの管理をするメイン処理関数、キー入力を見て自機の移動を行なう関数、自機を狙って弾を撃つ関数です。

このうち、後者の2つは通常のプレイとリプレイの区別をつけていません。その代わりランダムな要素が入っていない点に注意してください。

メイン処理

まず、メインの処理関数からです。ここでは管理のために3つのモードを切り替えています。

1つはキー入力を待つ、WAITモード、通常のプレイ中であるPLAYモード、リプレイ中であるREPLAYモードです。

最初にWAITモードから解説します。このモードでは、キー入力に合わせて、PLAYモード、またはREPLAYモードに移行します。

どちらのモードに移行しても、自機と弾を発射するタスクを作成します。

次にPLAYモードです。このモードではキー入力をバッファに記録し続けます。

この時同時に、自機の移動も並列処理で行なっているため、キー入力＝自機の移動パターンである事に注意してください。

入力は一定間取り続け、終了後、WAITモードに移ります。

最後はREPLAYモードです。

このモードではPLAYモードとは逆にバッファに記録したキー入力データを時間に合わせて読み

込み、キー入力バッファに反映しています。

自機の移動処理は、この入力を読み取るため、記録した時とまったく同じパターンで移動します。

同様に、自機を狙って弾を発射する処理も、まったく同じタイミングで自機を狙います。

このモードも一定時間再生を続けてあと終了し、WAIT モードに戻ります。

LIST 7 －16－1 リプレイの方法

```
#define REPLAY_COUNT 50*5
typedef struct{
    SPRITE          Sprt;
    SPRITE*         Target;                 //目標のスプライト
    int             Time;
    int             Time2;
    float           AddX;                   //増分値X
    float           AddY;                   //増分値Y
} EX07_16_BALL;

typedef struct{
    SPRITE          Sprt;
    int             Time;
} EX07_16_MYSHIP;

typedef struct{
    unsigned char   ReplayBuff[ REPLAY_COUNT ];
    int             Time;
    int             PlayMode;
} EX07_16_STRUCT;

#define MODE_WAIT     0
#define MODE_PLAY     1
#define MODE_REPLAY   2

unsigned char g_EX07_16_InputBuff;


void exec07_16_ball(TCB* thisTCB)
{
#define MOVE_SPEED  20.0
    EX07_16_BALL* work = (EX07_16_BALL*)thisTCB->Work;
    //自弾の座標
```

```
    float my_X;
    float my_Y;
    //目標の座標
    float targetX;
    float targetY;
    //目標の方向
    float direction;

    if( work->Time2++ > REPLAY_COUNT )
    {//一定時間処理を行なったら終了
        TaskKill( thisTCB );
    }else{
        //初期化処理、一定時間ごとに初期化される
        if( work->Time++ == 30 )
        {//一定時間ごとに座標と増分値を計算
            work->Sprt.X = SCREEN_WIDTH  / 2;
            work->Sprt.Y = SCREEN_HEIGHT / 2;
            work->Time = 0;

            //弾の座標を取得
            my_X    = work->Sprt.X;
            my_Y    = work->Sprt.Y;
            //目標の自機座標を取得
            targetX = work->Target->X + 16;
            targetY = work->Target->Y + 16;

            //座標から相手への方向を計算
            direction = atan2( targetY - my_Y, targetX - my_X );
            //方向から、X、Yそれぞれの座標増分値を計算
            work->AddX = cos( direction ) * MOVE_SPEED;
            work->AddY = sin( direction ) * MOVE_SPEED;
        }

        //座標に増分値を加算
        work->Sprt.X += work->AddX;
        work->Sprt.Y += work->AddY;

        SpriteDraw(&work->Sprt,1);
    }
}
```

```
void exec07_16_MyShip(TCB* thisTCB)
{
#define MOVE_SPEED  8.0
    EX07_16_MYSHIP* work = (EX07_16_MYSHIP*)thisTCB->Work;

    if( work->Time++ > REPLAY_COUNT )
    {//一定時間処理を行なったら終了
        TaskKill( thisTCB );
    }else{
        //リプレイ用にバッファに対応したキー入力による移動
        if( g_EX07_16_InputBuff & KEY_UP    ) work->Sprt.Y -= MOVE_SPEED;
        if( g_EX07_16_InputBuff & KEY_DOWN  ) work->Sprt.Y += MOVE_SPEED;
        if( g_EX07_16_InputBuff & KEY_RIGHT ) work->Sprt.X += MOVE_SPEED;
        if( g_EX07_16_InputBuff & KEY_LEFT  ) work->Sprt.X -= MOVE_SPEED;
        SpriteDraw( &work->Sprt, 0);
    }
}

void init07_16(TCB* thisTCB)
{
    //使用するテクスチャの読み込み
    D3DXCreateTextureFromFile( g_pD3Ddevice, "..¥¥..¥¥data¥¥0044.png",&g_pTex[0] );
    D3DXCreateTextureFromFile( g_pD3Ddevice, "..¥¥..¥¥data¥¥0055.png",&g_pTex[1] );
}

void exec07_16(TCB* thisTCB)
{
    EX07_16_STRUCT* work = (EX07_16_STRUCT*)thisTCB->Work;
    TCB*    myship_tcb;
    TCB*    ball_tcb;
    EX07_16_MYSHIP* myship_work;
    EX07_16_BALL* ball_work;
    RECT font_pos = { 0, 0, 640, 480, };
    char str[128];


    if( work->PlayMode == MODE_WAIT )
    {//入力待ち状態
        //Zキーでプレイモードに移行
```

```
            if( g_DownInputBuff & KEY_Z     ) work->PlayMode = MODE_PLAY;
            //Xキーでリプレイモードに移行
            if( g_DownInputBuff & KEY_X     ) work->PlayMode = MODE_REPLAY;

            //ボタンの入力があれば各処理を作成
            if( g_DownInputBuff & (KEY_Z | KEY_X) )
            {
                //自機のスプライト
                myship_tcb = TaskMake( exec07_16_MyShip, 0x2000 );
                myship_work = (EX07_16_MYSHIP*)myship_tcb->Work;
                myship_work->Sprt.X = SCREEN_WIDTH  / 2;
                myship_work->Sprt.Y = SCREEN_HEIGHT / 2 + 120;

                //ボールの初期化と、目標スプライトを設定
                ball_tcb = TaskMake( exec07_16_ball, 0x3000 );
                ball_work = (EX07_16_BALL*)ball_tcb->Work;
                ball_work->Target = &myship_work->Sprt;

            }
            //記録・再生用のタイマーを初期化
            work->Time = 0;
            g_pFont->DrawText( NULL, "Zキーで通常プレイ、Xキーでリプレイ", -1, &font_pos,
    DT_LEFT, 0xffffffff);
        }

        if( work->PlayMode == MODE_PLAY )
        {//通常プレイ、記録処理
            if( work->Time > REPLAY_COUNT )
            {//時間がきたらモードを切り替え記録終了
                work->PlayMode = MODE_WAIT;
            }else{
             //キー入力を記録
                g_EX07_16_InputBuff = g_InputBuff;
                work->ReplayBuff[ work->Time ] = g_EX07_16_InputBuff;
            }

            sprintf( str,"通常プレイ、キー入力記録中… %d",REPLAY_COUNT - work->Time);
            g_pFont->DrawText( NULL, str, -1, &font_pos, DT_LEFT, 0xffffffff);
```

```
        //時間を更新
        work->Time++;
    }

    if( work->PlayMode == MODE_REPLAY )
    {//リプレイ、再生処理
        if( work->Time > REPLAY_COUNT )
        {//時間がきたらモードを切り替え再生終了
            work->PlayMode = MODE_WAIT;
        }else{
         //記録したキー入力を再生
            g_EX07_16_InputBuff = work->ReplayBuff[ work->Time ];
        }

        sprintf( str,"リプレイ中… %d",REPLAY_COUNT - work->Time);
        g_pFont->DrawText( NULL, str, -1, &font_pos, DT_LEFT, 0xffffffff);

        //時間を更新
        work->Time++;
    }
}
```

7-17 レベルアップをするには

必要経験値の出し方

RPG等で、レベルアップを行なう方法を考えてみましょう。

一般的なレベルアップは、「経験値がレベルに合わせた一定値以上になったら、処理を行なう」というものでしょう。

ところが、この「レベルに合わせた」というのが曲者で、レベル1が100、レベル2が200、というように単純な計算式で表せればよいのですが、実際はゲームバランスなどから、単純な計算式を使う事は難しいでしょう。

また、複雑な計算式を使用した場合、調整の変更が面倒なだけでなく、バグの原因にもなりかねません。

そこで、通常は各レベルに対応した、「レベルアップデータ」を用いて処理を行ないます。

データ量は増えますが、こうする事で調整も容易になり、レベルアップしやすいキャラや、そうでないキャラなど、特徴的なレベルアップも可能になります。

レベルアップのプログラム

では、実際のプログラムを見ていきましょう。サンプルはZキーでランダムに経験値を加えていき、レベルアップの処理と表示を行なうプログラムです。

初期化とメイン処理

まずは初期化です。最初に現在のレベルと最初のレベルアップに必要な経験値を変数に設定します。

次にメインの処理を行ないます。まず、キー入力による経験値の取得処理です。

ここではランダムで0～50の経験値を取得するようにしています。

レベルアップの処理

次に、レベルアップの処理です。まず、現在の経験値が、レベルアップに必要な経験値に到達したかをチェックします。

最初に比較する経験値は初期化時に設定した物ですが、2回目のレベルアップからはレベルに合わせたデータを、レベルアップデータから取得して設定を行ないます。

そして、レベルアップ処理を行なうのですが、レベル最大時に経験値が半端な数だと余り見栄えが良くないため、ここでは経験値を0に戻す処理を加えています。

もちろん、この処理は無くても構いませんし、逆にレベル最大時に何らかの処理を行なうのであれば、ここで処理すると良いでしょう。

最後に、経験値とレベルの表示を行ない、この処理は終了です。

LIST 7－17－1 レベルアップをするには

```
#define LV_MAX 10
#define EXP_RANGE 50
typedef struct{
    int             NowEXP;         //現在の経験値
    int             NowLV;          //現在のレベル
    int             NextEXP;        //次にレベルアップする経験値
} EX07_17_STRUCT;

static int LevelUP_EXP[] =
{//レベルアップに必要な経験値のテーブル
    100,  //01
    100,  //02
    150,  //03
    200,  //04
    300,  //05
    450,  //06
    550,  //07
    600,  //08
    700,  //09
    800,  //10
    -1,   //MAX
};


void init07_17(TCB* thisTCB)
{
    EX07_17_STRUCT* work = (EX07_17_STRUCT*)thisTCB->Work;

    //レベルの初期化
    work->NowEXP = 0;
    work->NowLV  = 1;
```

```
        //最初のレベルアップに必要な経験値を取得、計算
        work->NextEXP += LevelUP_EXP[ work->NowLV ];
}

void exec07_17(TCB* thisTCB)
{
        EX07_17_STRUCT* work = (EX07_17_STRUCT*)thisTCB->Work;
        RECT font_pos = {   0,   0,640,480,};
        char str[128];

        //Zキーで経験値をランダムに加算
        if( g_DownInputBuff & KEY_Z )
        {//レベルが最大に達していたら、経験値を取得しないようにする
                if( work->NowLV < LV_MAX ) work->NowEXP += rand() % EXP_RANGE;
        }

        //レベルに対応した経験値を越えたらレベルアップ
        if( work->NowEXP >= work->NextEXP )
        {//レベルアップ処理
                //次のレベルアップに必要な経験値を取得、計算
                work->NextEXP += LevelUP_EXP[ work->NowLV ];
                //レベルアップ処理
                work->NowLV++;

                //レベルが最大時の処理
                if( work->NowLV >= LV_MAX )
                {
                        work->NowEXP  = 0;
                        work->NextEXP = 9999;
                }
        }
        //レベルと経験値の表示
        sprintf( str,"LEVEL:%2d¥n現在の経験値     :%8d¥n次LVまでの経験値:%8d",
                work->NowLV,work->NowEXP,work->NextEXP - work->NowEXP);
        g_pFont->DrawText( NULL, str, -1, &font_pos, DT_LEFT, 0xffffffff);

}
```

7-18 スコアランキングの管理

スコアランキング

スコアランキングの管理を行なってみましょう。

この処理は、シューティングをはじめとして、様々なゲームに実装されています。

ハイスコアを記録しておく事で、ゲーム攻略に対する目標が出来るため、ゲーム性の向上につながるからです。

また、ゲームセンターでは複数の人がゲームに挑戦するため、ハイスコアにランクインするかどうかで、自分の腕前を知る目安にもなります。

スコアランキング処理概要

早速処理手順を紹介していきます。

はじめに、ランクされているスコアに、プレイヤーが取得したスコアが、何位にランクイン出来るかを考えます。

ここは素直に考えて、取得したスコアを1位から順に比較していく事で、ランクインをチェックします。

この時、下位ランクから比較しても良いのですが、上位から比較する事でプログラムが若干楽になります。

比較中に、もし記録されているスコアが自分より低いスコアであれば、そこがランクの挿入位置になります。

もし最後まで、低いスコアが無ければランク外となり、ハイスコアには記録されない事になります。

スコアランキングのプログラム

初期化

では実際のプログラムを見ていきます。サンプルは、方向キーでスコアを増減させ、Zキーでそのスコアをランキングに登録するプログラムです。

まず、初期状態のスコアデータを用意します。ここでは1～5位のランクデータを処理するため、それに合わせたデータになっています。

また、スコアが順にならんでいる事をこのデータで保障する意味も含まれています。

メイン処理

次にメイン処理部分です。最初に、方向キーでスコアの増減を行ないます。この部分は、単純な加減算だけですので問題は無いでしょう。

スコアの登録

その後、キー入力によるスコアの登録処理を行ないます。

スコアの登録処理は、大きく2つに分けられます。すなわち、スコアがランクインされているかのチェックと、実際の登録処理です。

まず、関数EX07_18_rank_checkでランクインされているかをチェックします。この関数はスコアがランク外であれば負数を返します。

もし、チェック関数の返り値が正の数だった場合はランクインとなりますので、スコアの登録処理を行なっています。

その後、ランクキングの表示処理を行ない、メインの部分は終了です。

登録処理の関数

次に登録処理に関する関数を解説します。

まずチェック関数、EX07_18_rank_checkですが、これは先に説明したとおり、スコアを1位から順に比較していく関数です。

比較中、ランクインするようであれば、break文で抜け、その値を返します。もしランク外であれば−1を返します。

最後に登録関数、EX07_18_score_registです。

まず、処理のはじめにチェック関数を呼び出し、何位に登録するかをの順位を取得します。

その後、ランクを下位から順番に1つづつコピーしてずらしていきます。

これはランクの数だけ繰り返され、終了した時には挿入位置以降のランクは全て1位ずつ下がっています。

最後に、配列にスコアを登録して処理は終了です。

LIST 7 −18−1

```
#define RANK_COUNT 5

typedef struct{
    int             Score;
} EX07_18_STRUCT;

typedef struct{
    int             Score;
```

```
    char            Name[4];
} EX07_18_SCORE_STRUCT;

static EX07_18_SCORE_STRUCT gEX07_18_ScoreData[ RANK_COUNT ] =
{//初期スコアデータ
    { 10000,{'A','B','C','\0',},},
    {  5000,{'D','E','F','\0',},},
    {  3000,{'G','H','I','\0',},},
    {  1000,{'J','K','L','\0',},},
    {   500,{'M','N','O','\0',},},
};


int EX07_18_rank_check( int Score)
{
    int loop;

    //スコア登録、スコアの高い順にチェックしていく
    for( loop = 0; loop < RANK_COUNT; loop++ )
    {
        //もしランク内よりも高いスコアであれば、ランクイン
        if(Score > gEX07_18_ScoreData[ loop ].Score )break;
    }

    //ランクインかチェック
    if(loop != RANK_COUNT ) return loop;
    //ランク外
    return -1;
}


void EX07_18_score_regist( int Score, char* Name )
{
    int loop;
    int set_rank;

    //何位にランクインしているかを取得
    set_rank = EX07_18_rank_check( Score );

    for( loop = RANK_COUNT-1; loop > set_rank; loop-- )
```

```
        {//下位からランクを1つづつずらしていく
            gEX07_18_ScoreData[ loop ] = gEX07_18_ScoreData[ loop -1 ];
        }

        //スコアと名前を登録
        gEX07_18_ScoreData[ set_rank ].Score = Score;
        strcpy( gEX07_18_ScoreData[ set_rank ].Name, Name);
}


void exec07_18(TCB* thisTCB)
{
        EX07_18_STRUCT* work = (EX07_18_STRUCT*)thisTCB->Work;
        int loop;
        RECT font_pos = {   0,   0,640,480,};
        char str[128];

        //上キーでスコアを加算
        if( g_InputBuff & KEY_UP ) work->Score += 100;
        //下キーでスコアを減算
        if( g_InputBuff & KEY_DOWN )
        {
            work->Score -= 100;
            if( work->Score < 0 ) work->Score = 0;
        }
        //Zキーでスコア登録処理
        if( g_DownInputBuff & KEY_Z )
        {
            //ランクチェックを行い、ランクインならスコア登録処理
            if( 0 <= EX07_18_rank_check( work->Score ) )
            {
                EX07_18_score_regist( work->Score , "NEW" );
            }
        }

        //スコアデータの表示
        for( loop = 0; loop < RANK_COUNT; loop++ )
        {
            sprintf( str,
                "RANK%d: %8d  %s",
```

```
                loop+1,
                gEX07_18_ScoreData[ loop ].Score,
                gEX07_18_ScoreData[ loop ].Name);

            g_pFont->DrawText( NULL,
                str,
                -1,
                &font_pos,
                DT_LEFT,
                0xffffffff);

            font_pos.top += 32;
        }

        //現在のスコアの表示
        font_pos.top += 32;
        sprintf( str,"登録するスコア: %8d ",work->Score);
        g_pFont->DrawText( NULL,
            str,
            -1,
            &font_pos,
            DT_LEFT,
            0xffffffff);

        font_pos.top += 32;
        g_pFont->DrawText( NULL,
            "Zキーでスコア登録¥n方向キーでスコア変更",
            -1,
            &font_pos,
            DT_LEFT,
            0xffffffff);
    }
```

Chapter 8

当たり判定

8-1 点と円の当たり判定

ゲームと当たり判定

ゲームと当たり判定は、切っても切れない関係にあります。

シューティングをはじめ、アクションゲームやRPG、パズル等、ほとんどのゲームは当たり判定が無いと成り立たないといってもいいでしょう。

そんな当たり判定ですが、実はゲームやケースに応じて様々な種類があります。

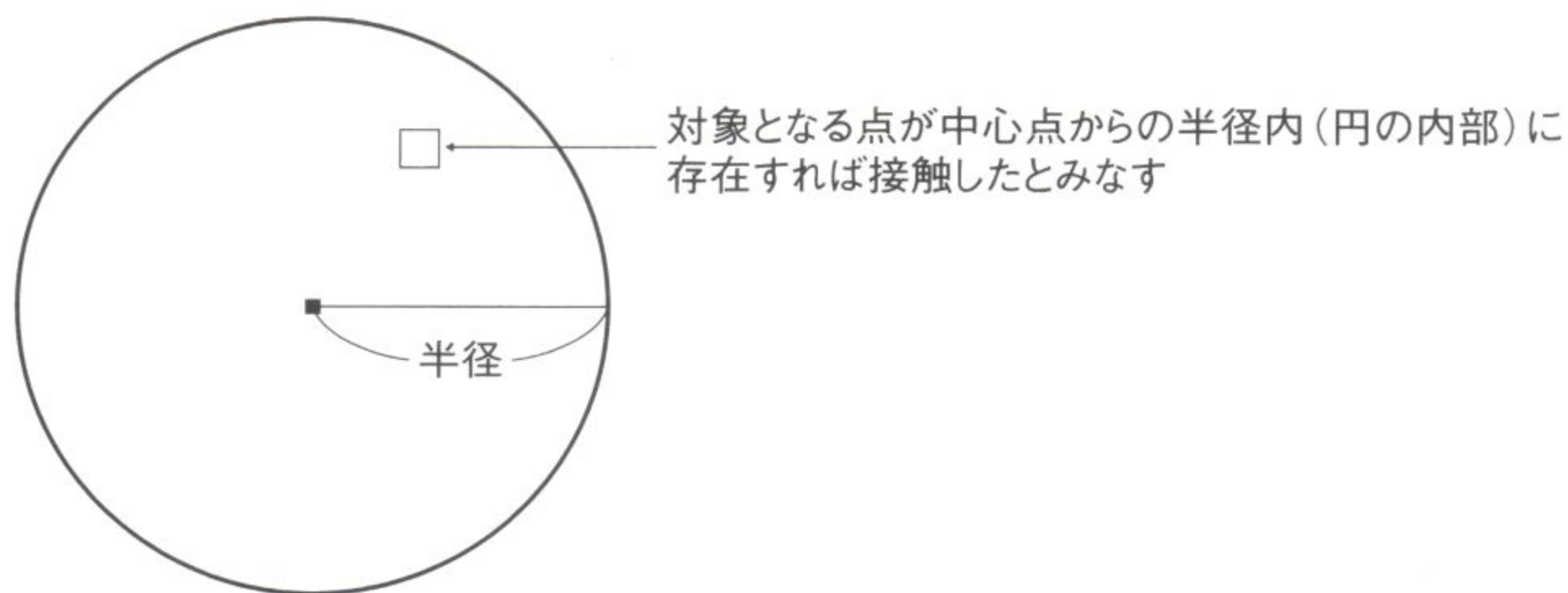
そのため全てを解説するのは困難ですが、本章ではその中で特に代表的な当たり範囲について解説します。

これらの当たり判定は、基本といってもよく、これさえ覚えておけば、2Dのに置ける大半のゲームはカバーできるでしょう。

円と点の当たり判定の概要

早速1つ目から行きましょう。まずは円と点との当たり判定です。

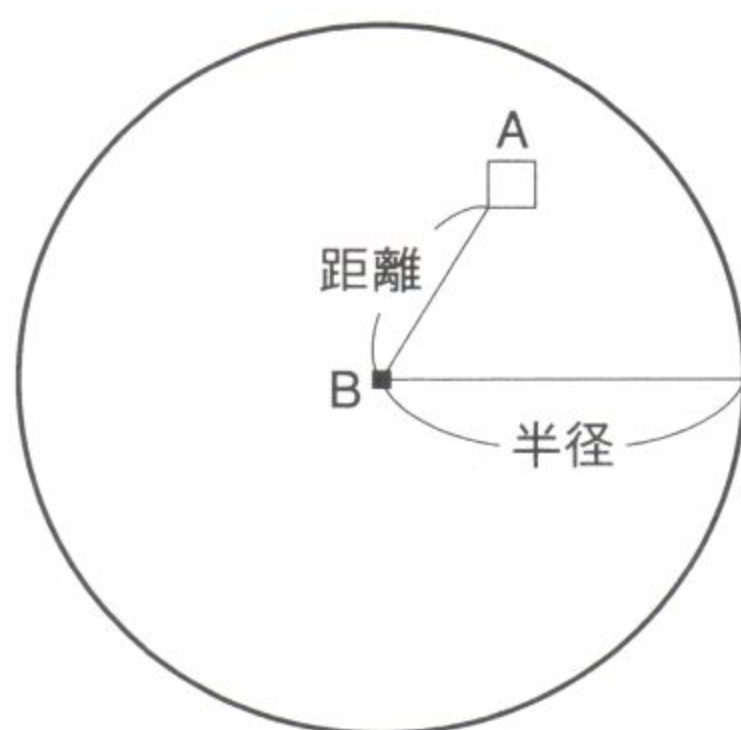
これはある一定の大きさを持った円と、点との当たり判定です。円の半径内に点が存在していれば当たっていると見なします。

図8－1－1 円と点とのイメージ図

ではどうやって、判定をするのでしょうか？　答えを先に書くと、判定には距離の公式を使います。

円の内部にあるということは、たとえどの位置にあっても、一定距離の範囲内にあります。

したがって点と点との距離を測り、その距離が一定範囲内にあれば、（＝円の範囲内にあれば）当たっているとみなす事が出来ます。

図8－1－2 円の内部にある点のイメージ図と2点間の距離の公式

A
距離
B
半径

対象となる点との距離と半径を
比較する事で判定を行なう

半径 ＞距離 なら接触

2点間の距離の公式

$$\sqrt{(AX - BX)^2 + (AY - BY)^2} \rightarrow \sqrt{(\text{点AのX} - \text{点BのX})^2 + (\text{点AのY} - \text{点BのY})^2}$$

円と点の当たり判定プログラム

では、実際のプログラムを見ていきましょう。

まず、自機の移動処理を行なっています。ここは特に注意点はありません。

次に、実際に距離を測る処理です。

ここは、上記の公式をそのままプログラムにすればOKです。自機の座標と目標の座標から距離を計算しています。

また、平方根を計算するために、関数sqrtfを呼び出しています※。

他に注意点として、目標の座標と自機の座標が同一だった場合、計算時にエラーが出てしまうので、距離を強制的に0にするようにしています。

最後に距離の表示と、自機、目標のスプライトの描画を行ないます。

なお、このサンプルでは、直接の当たり判定ではなく、距離の表示をしています。

サンプルを元に直接当たり判定に使用する際は、距離が代入される変数distanceを見ておき、一定距離以下かどうかで判断してください。

LIST 8－1－1 相手との距離を測る

```
typedef struct{
    SPRITE          MyShip;
    SPRITE          Target;
} EX08_01_STRUCT;

void init08_01(TCB* thisTCB)
```

※実際にはBCCにsqrtf関数はありません。DirectXでも使用できるように本書のプログラムではマクロで定義されています。

```
{
    EX08_01_STRUCT* work = (EX08_01_STRUCT*)thisTCB->Work;

    //使用するテクスチャの読み込み
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥0044.png",&g_pTex[0] );
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥0055.png",&g_pTex[1] );

    //目標の初期座標
    work->Target.X = SCREEN_WIDTH  / 2;
    work->Target.Y = SCREEN_HEIGHT / 2;
}

void exec08_01(TCB* thisTCB)
{
#define MOVE_SPEED  8.0

    EX08_01_STRUCT* work = (EX08_01_STRUCT*)thisTCB->Work;
    float distance;
    char str[128];
    RECT font_pos = { 0, 0, 640, 480, };

    //キー入力による移動
    if( g_InputBuff & KEY_UP    ) work->MyShip.Y -= MOVE_SPEED;
    if( g_InputBuff & KEY_DOWN  ) work->MyShip.Y += MOVE_SPEED;
    if( g_InputBuff & KEY_RIGHT ) work->MyShip.X += MOVE_SPEED;
    if( g_InputBuff & KEY_LEFT  ) work->MyShip.X -= MOVE_SPEED;


    //同一座標の場合エラーが出るので特別処理
    if( (work->Target.X == work->MyShip.X )&&(work->Target.Y == work-
>MyShip.Y) )
    {
        distance = 0;
    }else{
        //相手との距離を計測
        distance = sqrtf( (work->Target.X - work->MyShip.X) * (work-
>Target.X - work->MyShip.X) +
                          (work->Target.Y - work->MyShip.Y) * (work-
```

```
>Target.Y - work->MyShip.Y)
                                );
        }
        sprintf( str,"距離  %f",distance);
        g_pFont->DrawText( NULL, str, -1, &font_pos, DT_LEFT, 0xffffffff);

        SpriteDraw(&work->MyShip,0);
        SpriteDraw(&work->Target,1);

}
```

8-2 円同士の当たり判定

円同士の当たり判定の概要

次に２つ目として、円同士の当たり判定を考えてみましょう。

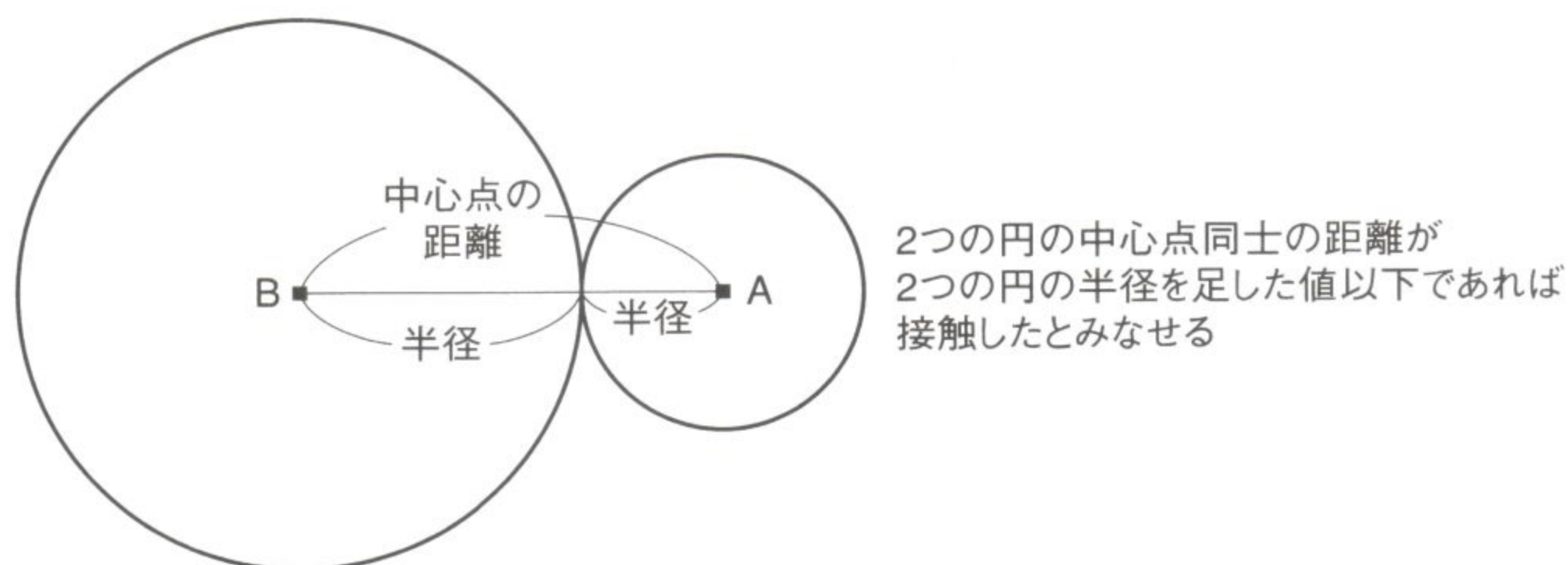
これはある一定の大きさを持った円と、違う大きさを持った円との当たり判定です。

円同士が接触していたら、当たっているとみなします。これは、色々な大きさの物体同士で当たり判定を持たせたい時に有効です。

判定方法

判定方法ですが、実はこちらも、距離の公式を使って判定する事が出来ます。

２点間の距離を計算したら、その距離が、当たり判定を行なう２つの円をの半径を足した値以下になっていれば、当たっているとみなす事が出来ます。

図8－2－1 円と円との当たり判定処理のイメージ図

当たり判定プログラムの高速化

実際にプログラムを見てもらえれば分かりますが、この処理はほとんど点と円の当たり判定と同じです。

このままではなんなので、少し高速化を行なってみましょう。

このアルゴリズムでは、距離を求めるために平方根を使っていますが、実は一定の範囲内にあるかどうかをチェックするだけなら、わざわざ平方根を求める必要はありません。

２点間が接触する距離の２乗を計算して判定値として記録しておき、その判定値より大きいか小さいかで距離の大小関係を判定します。

これは平方根の計算を行なう前後でも、数値の大小関係は変わらない事を利用しています。

図８ −２−２ 平方根の計算の前後でも、数値の大小関係が変わらない計算のイメージ図

2点間の距離の公式

平方根の計算の前後でも関係は変わらない

$$\sqrt{(AX - BX)^2 + (AY - BY)^2} = D \longrightarrow (AX - BX)^2 + (AY - BY)^2 = D^2$$

距離 ＝ D として両辺を2 乗する

円同士の当たり判定プログラム

高速化

高速化の部分以外はほとんど同じですので、そこの部分に絞って解説します。

まず、実際の座標から判定のための値を求める部分ですが、平方根を求める関数を無くしています。

また関数が無くなった事により、距離が同一の場合でもエラーが出なくなったので、そのための処理も無くしています。

次に判定値を求める部分ですが、それぞれの当たり判定の大きさを２乗して足しています。

ここは本来、２つの当たり判定の大きさを求めて距離を計算し、２乗するのが正しいのですが、まったく同じ計算を逆にする事になってしまうので、省略化を行なっています。

判定値を求めたら、後は大小関係をチェックするだけです。

チェックする数が１つだけでは余り高速化の恩恵は感じられませんが、実際のゲームでは多数のチェックをする事が多くなります。

そんな時には、この手法を試してみてください。

LIST 8 −２−１ 円同士の当たり判定高速化

```
	//接触のための判定値を計算
	distance = (work->Target.X - work->MyShip.X) * (work->Target.X - work->MyShip.X) +
	           (work->Target.Y - work->MyShip.Y) * (work->Target.Y - work->MyShip.Y) ;
```

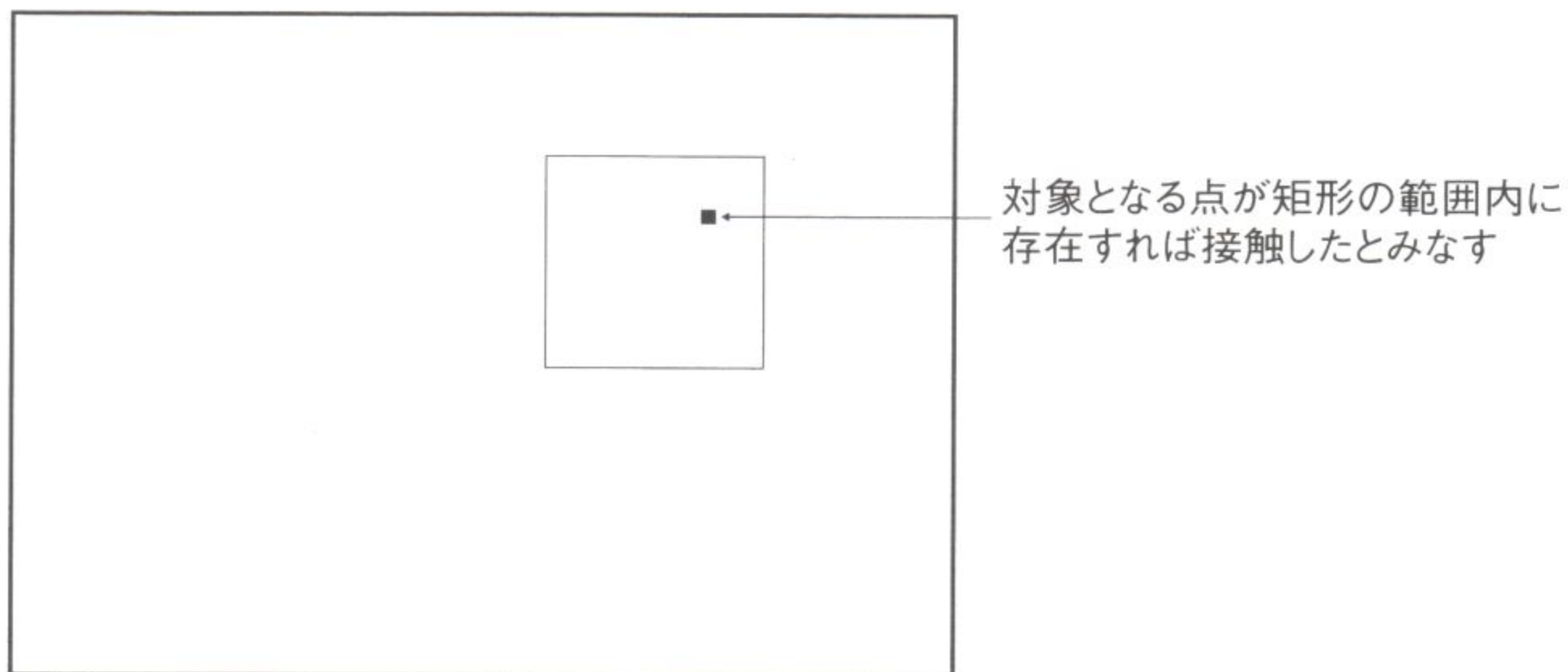
```
    //判定値として距離の二乗を求めるが、同じ計算なので省略
    //check_distance =
    //    pow( sqrtf( ((HIT_SIZE1 - 0) * (HIT_SIZE1 - 0)) + ((HIT_SIZE2 - 0)
 * (HIT_SIZE2 - 0)) ) );
    check_distance = (HIT_SIZE1 * HIT_SIZE1) + (HIT_SIZE2 * HIT_SIZE2) ;
    //判定値以下なら接触
    if( distance < check_distance )
    {
        sprintf( str,"接触しました");
        g_pFont->DrawText( NULL, str, -1, &font_pos, DT_LEFT, 0xffffffff);
    }
```

8-3 矩形と点の当たり判定

矩形と点の当たり判定の概要

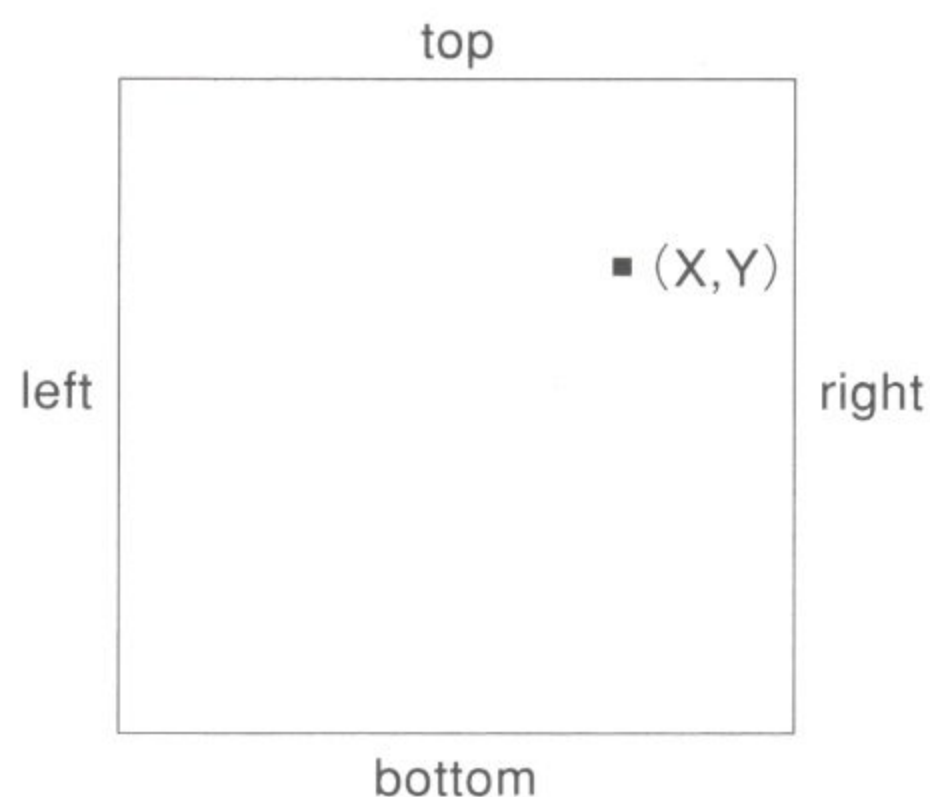
次は、矩形と点の当たり判定を考えてみましょう。

これは任意の大きさを持った矩形と点との当たり判定です。

図8 －3－1 矩形と点との当たり判定のイメージ図

この当たり判定の特徴はとにかく単純で、高速な事です。

また、実際の適用範囲も広く、この当たり判定だけで処理が済んでしまうゲームもあります。

判定方法ですが、矩形を構成する4つの座標と点を比較して、座標の範囲内に点が入っていれば、当たっているとみなします。

図8 －3－2 4つの座標の範囲内に点があれば当たっている判定のイメージ図

矩形と点の当たり判定のプログラム

実際の判定部分のプログラムを見ていきましょう。

まず、当たり判定の大きさと自機の座標から、判定のための矩形を計算します。
次に目標の座標が、計算された座標の内側にあるかを１つづつチェックします。
もし全ての座標で内側にあれば、当たっていると判断します。
アルゴリズムが非常に単純なため、特に注意する事はありません。

ただ、この判定処理はゲーム中以外でも使う事がありますので、もし頻繁に使うようであれば関数化しておくと良いでしょう。

LIST 8 ー3ー1 矩形と点の当たり判定

```
typedef struct{
    SPRITE          MyShip;
    SPRITE          Target;
} EX08_03_STRUCT;

void init08_03(TCB* thisTCB)
{
    EX08_03_STRUCT* work = (EX08_03_STRUCT*)thisTCB->Work;

    //使用するテクスチャの読み込み
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥0044.png",&g_pTex[0] );
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥0055.png",&g_pTex[1] );

    //目標の初期座標
    work->Target.X = SCREEN_WIDTH  / 2;
    work->Target.Y = SCREEN_HEIGHT / 2;
}

void exec08_03(TCB* thisTCB)
{
#define MOVE_SPEED  8.0

    EX08_03_STRUCT* work = (EX08_03_STRUCT*)thisTCB->Work;
```

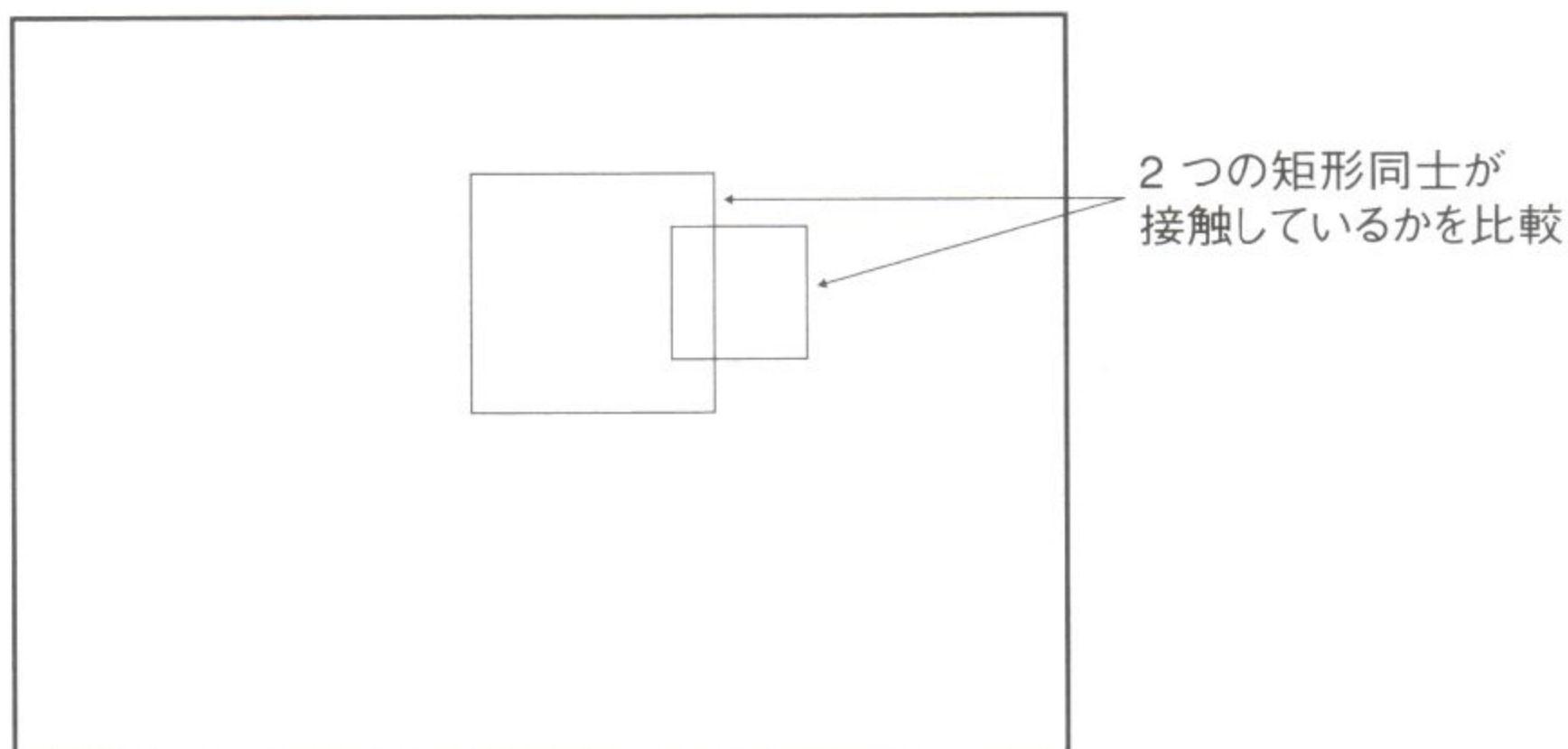
```
    char str[128];
    RECT hit_size_rect = { 0, 0, 64, 64, };
    RECT hit_rect;
    RECT font_pos = { 0, 0, 640, 480, };

    //キー入力による移動
    if( g_InputBuff & KEY_UP    ) work->MyShip.Y -= MOVE_SPEED;
    if( g_InputBuff & KEY_DOWN  ) work->MyShip.Y += MOVE_SPEED;
    if( g_InputBuff & KEY_RIGHT ) work->MyShip.X += MOVE_SPEED;
    if( g_InputBuff & KEY_LEFT  ) work->MyShip.X -= MOVE_SPEED;


    //判定チェックのための矩形を計算
    hit_rect.top    =   work->MyShip.Y + hit_size_rect.top;
    hit_rect.bottom =   work->MyShip.Y + hit_size_rect.bottom;
    hit_rect.left   =   work->MyShip.X + hit_size_rect.left;
    hit_rect.right  =   work->MyShip.X + hit_size_rect.right;

    //対象が矩形の範囲内にあれば当たっている
    if( hit_rect.top    < work->Target.Y  &&
        hit_rect.bottom > work->Target.Y  &&
        hit_rect.left   < work->Target.X  &&
        hit_rect.right  > work->Target.X  )
    {
        sprintf( str,"接触しました");
        g_pFont->DrawText( NULL, str, -1, &font_pos, DT_LEFT, 0xffffffff);
    }

    SpriteDraw(&work->MyShip,0);
    SpriteDraw(&work->Target,1);

}
```

8-4 矩形同士の当たり判定

矩形同士の当たり判定の概要

最後に、矩形同士の当たり判定を考えてみましょう。

これは任意の大きさを持った矩形同士の当たり判定です。

図8－4－1 矩形同士の当たり判定のイメージ図

この当たり判定の特徴は、単純である事、また矩形情報を複数持って組み合わせれば、複雑な形状同士の判定も行なえる事です。

そのため、格闘ゲームを始めとした複雑な当たり判定を必要とするゲームで広く使われます。

どうやって判定するか

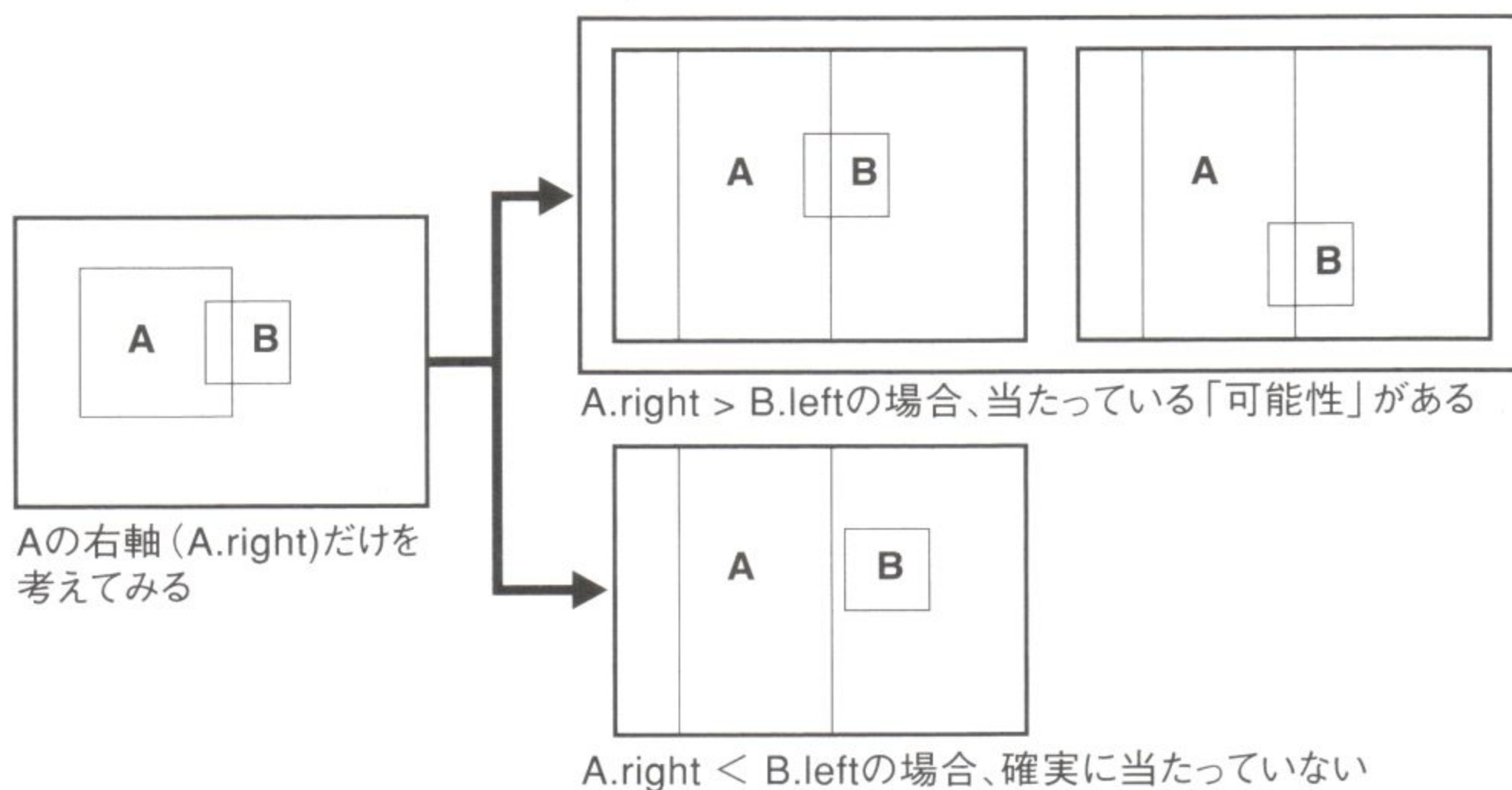
判定方法ですが、若干複雑です。

まず、対象となる２つの矩形を上下左右４つの枠に分解した状態をイメージしてください。

この際、判定元の矩形の上下左右の枠が、対象となる矩形の対の枠（上なら下の枠、左なら右の枠）の内側に含まれて「いない」か？　をチェックします。

もし、含まれていなければ、当たっていませんが、逆に一部でも含まれていたら、当たっている可能性があります。

これを上下左右の枠全てについてチェックし、もし全ての枠が当たっていれば、それは当たっているとみなす事が出来ます。

図8－4－2 矩形を構成する上下左右の各軸が対象となる矩形の範囲を超えている際のイメージ図

この判定を上下左右4 つの軸に対して行い、全ての辺が「当たっている可能性がある」のであれば、当たっている

矩形同士の当たり判定のプログラム

では、実際の判定部分のプログラムを見ていきましょう。

判定のイメージは若干難しいですが、プログラムにすると、とても単純になります。

まず、当たり判定の大きさと自機、及び目標の座標から、判定のための矩形を計算します。

次に算出した矩形同士を比較します。この際の比較は上記の通り、対となる枠に対しての比較です。

対になるのは、上枠⇔下枠、左枠⇔右枠で、対象枠の内側に判定枠が存在するかを比較しています。

判定部分のプログラムは非常に単純ですので、上記の解説でイメージがつかみにくい様であれば、プログラムと見比べながら理解してください。

LIST 8－4－1 矩形同士の当たり判定

```
typedef struct{
    SPRITE          MyShip;
    SPRITE          Target;
} EX08_04_STRUCT;

void init08_04(TCB* thisTCB)
{
    EX08_04_STRUCT* work = (EX08_04_STRUCT*)thisTCB->Work;
```

```
	//使用するテクスチャの読み込み
	D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥0044.png",&g_pTex[0] );
	D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥0055.png",&g_pTex[1] );

	//目標の初期座標
	work->Target.X = SCREEN_WIDTH  / 2;
	work->Target.Y = SCREEN_HEIGHT / 2;
}

void exec08_04(TCB* thisTCB)
{
#define MOVE_SPEED  8.0

	EX08_04_STRUCT* work = (EX08_04_STRUCT*)thisTCB->Work;
	char str[128];
	RECT hit_size_rectA = { 0, 0, 64, 64, };
	RECT hit_size_rectB = { 0, 0, 32, 32, };
	RECT hit_rectA;
	RECT hit_rectB;
	RECT font_pos = { 0, 0, 640, 480, };

	//キー入力による移動
	if( g_InputBuff & KEY_UP    ) work->MyShip.Y -= MOVE_SPEED;
	if( g_InputBuff & KEY_DOWN  ) work->MyShip.Y += MOVE_SPEED;
	if( g_InputBuff & KEY_RIGHT ) work->MyShip.X += MOVE_SPEED;
	if( g_InputBuff & KEY_LEFT  ) work->MyShip.X -= MOVE_SPEED;


	//判定チェックのための自機の当たり判定矩形Aを計算
	hit_rectA.top    =   work->MyShip.Y + hit_size_rectA.top;
	hit_rectA.bottom =   work->MyShip.Y + hit_size_rectA.bottom;
	hit_rectA.left   =   work->MyShip.X + hit_size_rectA.left;
	hit_rectA.right  =   work->MyShip.X + hit_size_rectA.right;

	//判定チェックのための目標の当たり判定矩形Bを計算
	hit_rectB.top    =   work->Target.Y + hit_size_rectB.top;
	hit_rectB.bottom =   work->Target.Y + hit_size_rectB.bottom;
```

```
    hit_rectB.left    =   work->Target.X + hit_size_rectB.left;
    hit_rectB.right   =   work->Target.X + hit_size_rectB.right;

    //対象枠全てが対となる矩形の範囲内にあれば当たっている
    if( hit_rectA.top    < hit_rectB.bottom &&
        hit_rectA.bottom > hit_rectB.top    &&
        hit_rectA.left   < hit_rectB.right  &&
        hit_rectA.right  > hit_rectB.left   )
    {
        sprintf( str,"接触しました");
        g_pFont->DrawText( NULL, str, -1, &font_pos, DT_LEFT, 0xffffffff);
    }

    SpriteDraw(&work->MyShip,0);
    SpriteDraw(&work->Target,1);
}
```

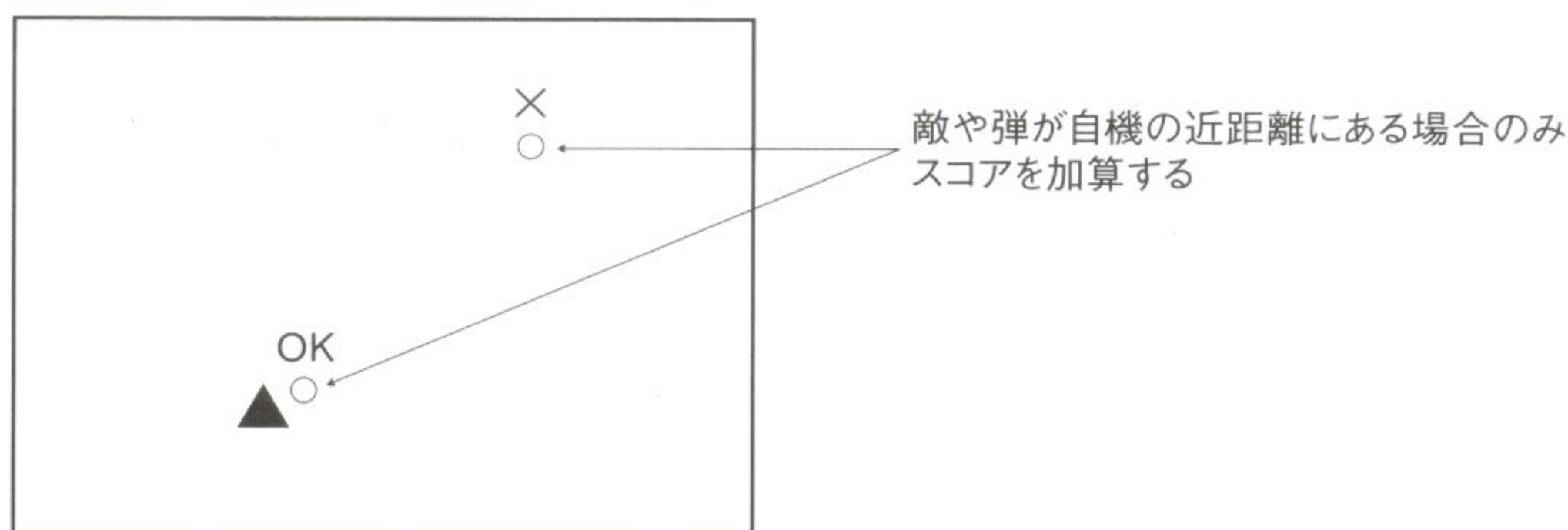
8-5 自機に弾をかすらせて得点

弾をかすらせるとは？

最近のゲームでは「かすり点」というシステムが導入されているものがあります。

これはどういう物かと言うと、自機が敵の弾等に非常に接近した場合に、スコア等にボーナスが得られるシステムの事です。

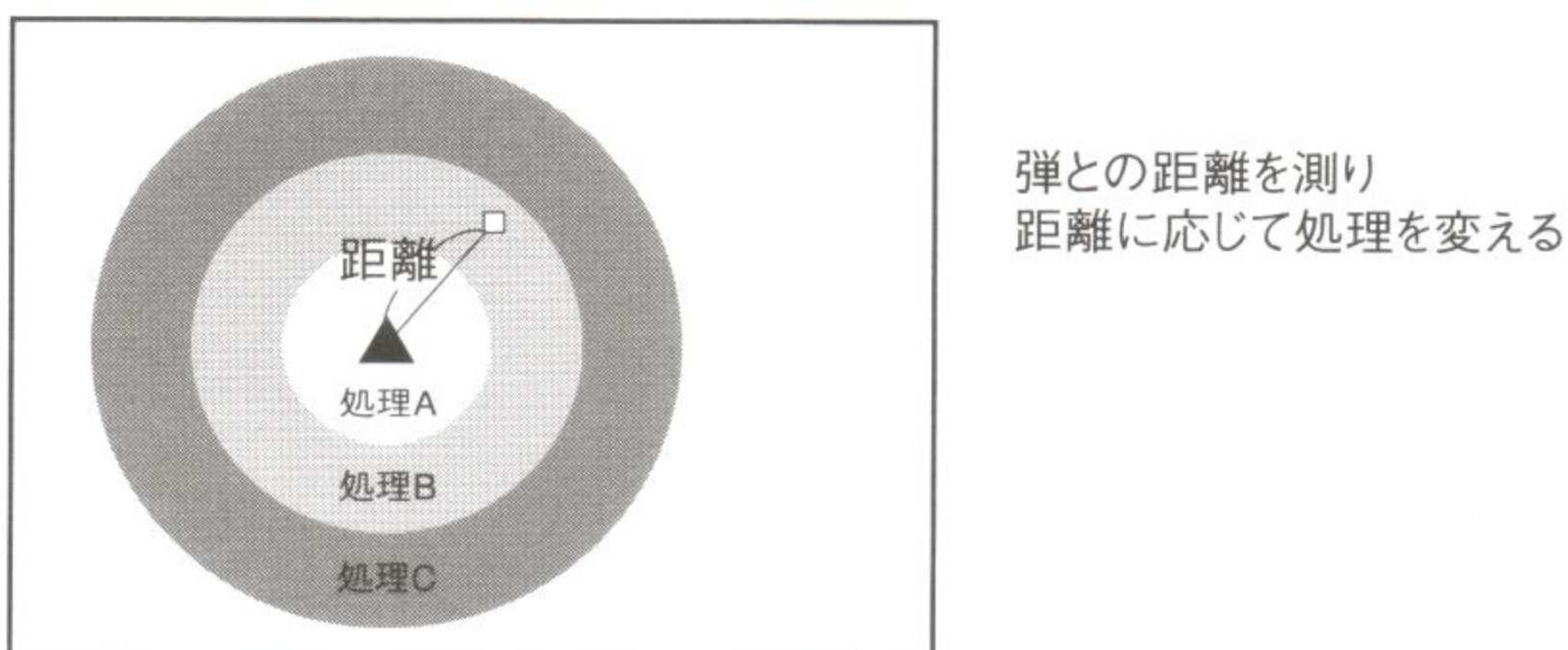
上級者ほど、弾に近づいてスコアを得るようになるため、緊張感が増し、ゲーム性が高くなります。

図8－5－1 敵や弾に近づくとスコアが加算されるシステムのイメージ図

かすらせて得点する処理の考え方

それではこのようなシステムは、どのようにして実現しているのでしょうか？

実は仕組みは単純で、点と円の当たり判定の項目で、２点間の距離の公式を使用します。

計算で算出した距離を当たり判定だけに用いるのではなく、距離に応じて処理を変えてやればこのシステムは実現できます。

図8－5－2 距離に応じて処理を変更するイメージ図

ここまで分かれば、実際のプログラムは難しくありません。

かすらせて得点するプログラム

サンプルプログラムを示します。

処理は円と点の当たり判定のプログラムとほとんど同じで、距離に応じて処理を振り分けています。

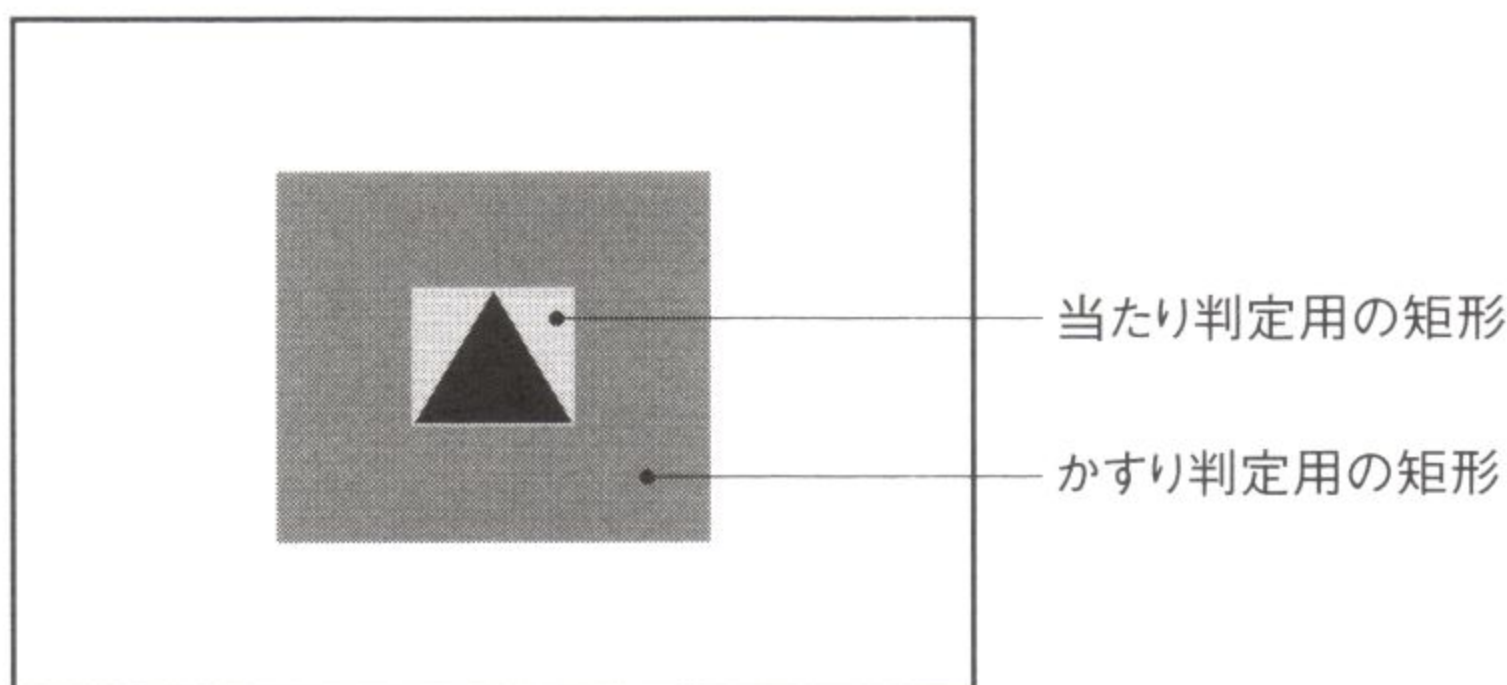
サンプルでは、上から飛んでくる弾にかするとスコアが入り、命中するとスコアが0になる、簡単なミニゲームになっています。

なお、ここでは、円との当たり判定を利用していますが、矩形の当たり判定でもかすりシステムは実現できます。

その場合は、かすり用と当たり判定用の、大きさの違う2つの矩形を持ち、それぞれの矩形に対して処理を行なうようにすると良いでしょう。

図8－5－3　2種類の矩形をもち、それに合わせて処理を変更するイメージ図

当たり判定用の矩形

かすり判定用の矩形

LIST 8－5－1 弾とかすってスコアを得る

```
typedef struct{
    SPRITE          MyShip;
    SPRITE          Target;
    int             Score;
} EX08_05_STRUCT;

void init08_05(TCB* thisTCB)
{
    EX08_05_STRUCT* work = (EX08_05_STRUCT*)thisTCB->Work;

```

```
	//使用するテクスチャの読み込み
	D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥0044.png",&g_pTex[0] );
	D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥0055.png",&g_pTex[1] );

	//目標の初期座標
	work->Target.X = SCREEN_WIDTH  / 2;
	work->Target.Y = SCREEN_HEIGHT / 2;
}

void exec08_05(TCB* thisTCB)
{
#define MOVE_SPEED    8.0
#define TARGET_SPEED  10.0
#define HIT_SIZE      16.0        //当たり判定の大きさ
#define GRAZE_DISTANCE  64.0     //かすりの距離

	EX08_05_STRUCT* work = (EX08_05_STRUCT*)thisTCB->Work;
	float distance;
	char str[128];
	RECT font_pos = { 0, 0, 640, 480, };

	//キー入力による移動
	if( g_InputBuff & KEY_UP    ) work->MyShip.Y -= MOVE_SPEED;
	if( g_InputBuff & KEY_DOWN  ) work->MyShip.Y += MOVE_SPEED;
	if( g_InputBuff & KEY_RIGHT ) work->MyShip.X += MOVE_SPEED;
	if( g_InputBuff & KEY_LEFT  ) work->MyShip.X -= MOVE_SPEED;


	//目標の移動
	work->Target.Y += TARGET_SPEED;
	if( work->Target.Y > SCREEN_HEIGHT)
	{
		work->Target.Y = 0;
		work->Target.X = SCREEN_WIDTH/2;
	}
	//同一座標の場合エラーが出るので特別処理
	if( (work->Target.X == work->MyShip.X )&&
		(work->Target.Y == work->MyShip.Y) )
```

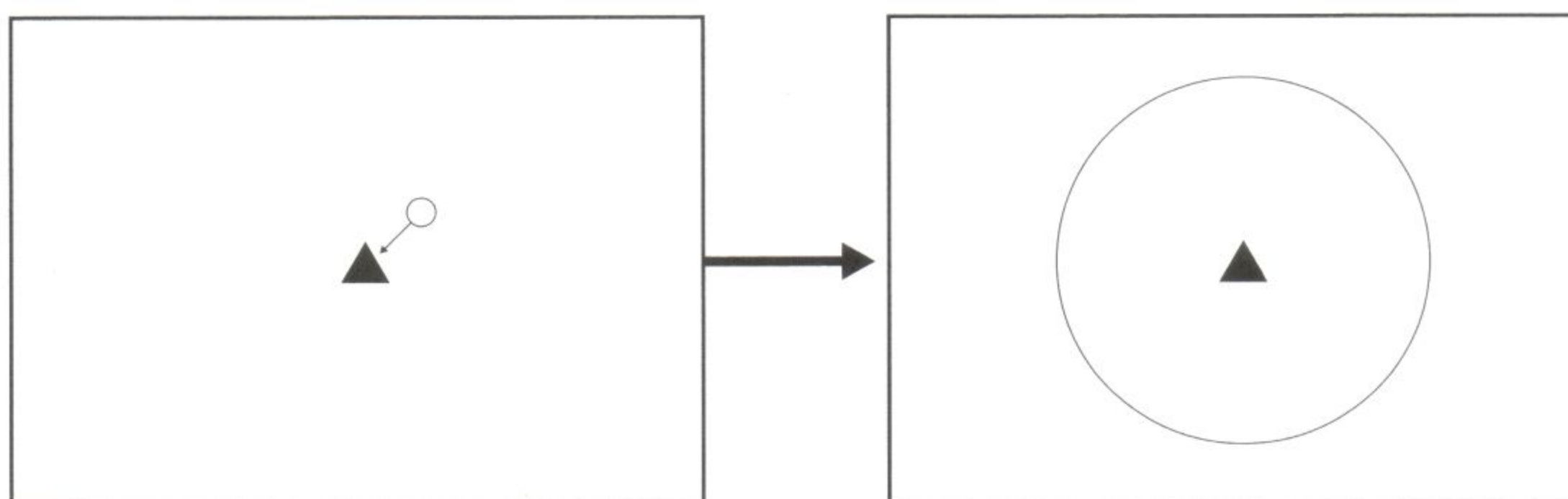
```
    {
        distance = 0;
    }else{
        //相手との距離を計測
        distance = sqrtf( (work->Target.X-8 - work->MyShip.X-16) *
                          (work->Target.X-8 - work->MyShip.X-16) +
                          (work->Target.Y-8 - work->MyShip.Y-16) *
                          (work->Target.Y-8 - work->MyShip.Y-16)
                        );
    }

    //距離が一定以内なら掠っていると判定、スコアを加算
    if(distance <= GRAZE_DISTANCE)work->Score += 1;

    //接触したらスコアをクリア
    if(distance <= HIT_SIZE)work->Score = 0;


    sprintf( str,"スコア  %d", work->Score );
    g_pFont->DrawText( NULL, str, -1, &font_pos, DT_LEFT, 0xffffffff);

    SpriteDraw(&work->MyShip,0);
    SpriteDraw(&work->Target,1);
}
```

8-6 回避ボム

緊急回避用システム

シューティングゲームでよく使われる緊急回避ボムを作成してみましょう。

緊急回避ボムとは、ゲームの初心者向けに考えられたシステムで、弾が自機に命中する、あるいは命中しそうになった場合、自動的にボムを発射して弾を回避するシステムの事です。

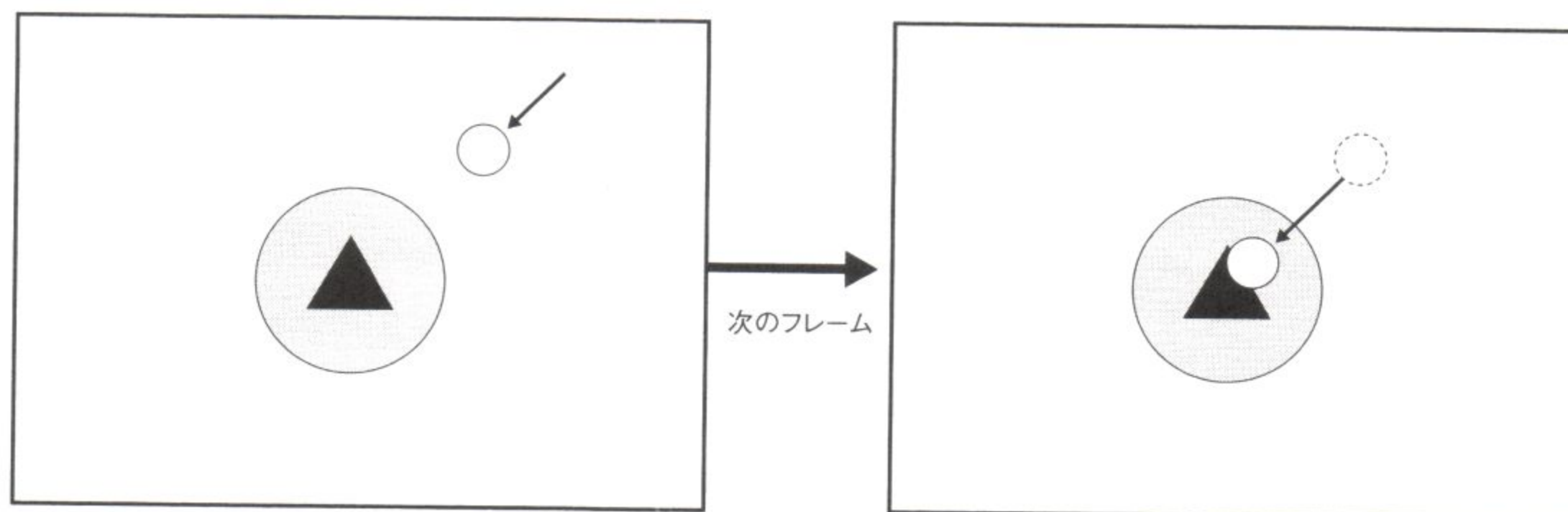
図8－6－1 回避ボムのイメージ図

自機に弾が当たる瞬間にボムを発射する

回避ボムの種類

この回避ボムのパターンですが、前述したように２種類のアルゴリズムがあります。

１つは弾が自機に接触した時にボムが発射されるタイプ、もう１つは、弾が自機にある程度近づいた時に発射されるタイプです。

基本的にはどちらでもかまわないのですが、後者の場合、高速で弾が飛来した時に、ボムが発動する前に自機が死んでしまう場合があります。

図8－6－2 近接時にボム発動のアルゴリズムの場合、死亡するケースのイメージ図

高速で弾が移動している場合近接判定のチェックをすり抜けて直接自機に当たる場合がある

前者のアルゴリズムの場合ですと、接触時にボムが発動するため、この現象は起りません。

回避ボムを当てにしてゲームをプレイして欲しく無い場合などに、後者のアルゴリズムは有効ですので、ゲーム内容やゲームバランスに応じてアルゴリズムを選択すると良いでしょう。

回避ボムのプログラム

さて、実際のリストです。サンプルでは、自機に接触した時にボムを発動するようにしています。

弾との接触時に、通常ですと死亡判定のプログラムを呼び出すのですが、ここではその代わりに、ボムを発射するようにしています。

LIST 8－6－1 回避ボム

```
#define BOMB_MAX 3
#define BOMB_TIME 50

typedef struct{
    SPRITE          MyShip;
    SPRITE          Target;
    SPRITE          Bomb;
    int             BombCount;
    int             BombFlag;
    int             BombTime;
} EX08_06_STRUCT;

void init08_06(TCB* thisTCB)
{
    EX08_06_STRUCT* work = (EX08_06_STRUCT*)thisTCB->Work;

    //使用するテクスチャの読み込み
```

```
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥0044.png",&g_pTex[0] );
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥0055.png",&g_pTex[1] );
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥spread.png",&g_pTex[2] );

    //目標の初期座標
    work->Target.X = SCREEN_WIDTH  / 2;
    work->Target.Y = 0;
    work->BombCount = BOMB_MAX;
}

void exec08_06(TCB* thisTCB)
{
#define MOVE_SPEED    8.0
#define TARGET_SPEED  10.0

    EX08_06_STRUCT* work = (EX08_06_STRUCT*)thisTCB->Work;
    RECT hit_size_rectA = { 0, 0, 64, 64, };
    RECT hit_size_rectB = { 0, 0, 32, 32, };
    RECT hit_rectA;
    RECT hit_rectB;

    char str[128];
    RECT font_pos = { 0, 0, 640, 480, };
    RECT bomb_data ={192,  0,256, 64, };


    //キー入力による移動
    if( g_InputBuff & KEY_UP    ) work->MyShip.Y -= MOVE_SPEED;
    if( g_InputBuff & KEY_DOWN  ) work->MyShip.Y += MOVE_SPEED;
    if( g_InputBuff & KEY_RIGHT ) work->MyShip.X += MOVE_SPEED;
    if( g_InputBuff & KEY_LEFT  ) work->MyShip.X -= MOVE_SPEED;

    //目標の移動
    work->Target.Y += TARGET_SPEED;
    if( work->Target.Y > SCREEN_HEIGHT)
    {
        work->Target.Y = 0;
```

```
        work->Target.X = SCREEN_WIDTH/2;
    }


    if( g_DownInputBuff & KEY_X  )
    {//Xキーでボム数回復
        work->BombCount = BOMB_MAX;
    }

    if(!work->BombFlag)
    {//ボムが発動していない時だけ処理
        //Zキーで爆発処理
        if( g_DownInputBuff & KEY_Z  )
        {//ここでは、ボムの発動と、ボム数の計算を行なう
            if(work->BombCount > 0)
            {//ただし、ボムがない場合のみ
                work->BombFlag = true;
                if(--work->BombCount < 0) work->BombCount = 0;
                //ボムへ座標をコピー
                work->Bomb.X = work->MyShip.X;
                work->Bomb.Y = work->MyShip.Y;
            }
        }


        //判定チェックのための自機の当たり判定矩形Aを計算
        hit_rectA.top     =   work->MyShip.Y + hit_size_rectA.top;
        hit_rectA.bottom =   work->MyShip.Y + hit_size_rectA.bottom;
        hit_rectA.left    =   work->MyShip.X + hit_size_rectA.left;
        hit_rectA.right   =   work->MyShip.X + hit_size_rectA.right;

        //判定チェックのための目標の当たり判定矩形Bを計算
        hit_rectB.top     =   work->Target.Y + hit_size_rectB.top;
        hit_rectB.bottom =   work->Target.Y + hit_size_rectB.bottom;
        hit_rectB.left    =   work->Target.X + hit_size_rectB.left;
        hit_rectB.right   =   work->Target.X + hit_size_rectB.right;

        //弾に当たった場合、回避処理としてボムの発動
        if( hit_rectA.top    < hit_rectB.bottom &&
            hit_rectA.bottom > hit_rectB.top    &&
```

```
        hit_rectA.left    < hit_rectB.right  &&
        hit_rectA.right   > hit_rectB.left    )
    {//ボタン押しと同様に、発動処理と計算を行なう

        //ただし、ボムがない場合は命中のメッセージを表示
        if(work->BombCount <= 0)
        {
            sprintf( str,"接触しました");
            g_pFont->DrawText( NULL,
                           str, -1, &font_pos,
                            DT_LEFT, 0xffffffff);
        }
        if(work->BombCount > 0)
        {
            work->BombFlag = true;
            //ボムへ座標をコピー
            work->Bomb.X = work->MyShip.X;
            work->Bomb.Y = work->MyShip.Y;

            if(--work->BombCount < 0) work->BombCount = 0;
        }
    }
}


if(work->BombFlag)
{//ボム発動時の処理
    //ボムの表示
    work->Bomb.SrcRect = &bomb_data;

    SpriteDraw(&work->Bomb,2);

    //ボムの発動時間を進める
    work->BombTime++;
    //指定の時間に達していれば終了
    if(work->BombTime >= BOMB_TIME)
    {//終了処理
        work->BombTime = 0;
        work->BombFlag = false;
    }
```

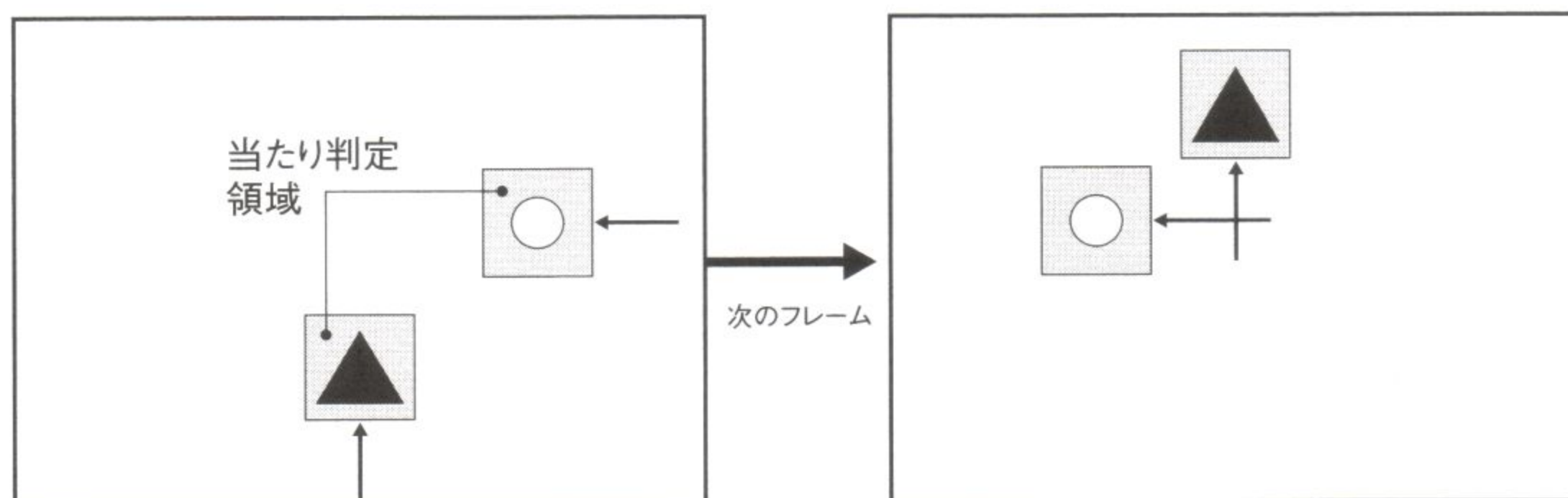
```
    }




    font_pos.top +=16 ;
    sprintf( str,"残ボム数 = %d",work->BombCount);
    g_pFont->DrawText( NULL, str, -1, &font_pos, DT_LEFT, 0xffffffff);

    SpriteDraw(&work->MyShip,0);
    SpriteDraw(&work->Target,1);
}
```

8-7 スピードアップ時の当たり判定

当たり判定のすり抜け

スピードアップ時の当たり判定について考えてみます。
通常、当たり判定は現在のフレームについてのみ判断を行ないます。
普通は、これで問題が出ることはありません。

しかし、あまりに高速で物体を移動させたり、極端に当たり判定が小さいと、移動範囲が当たり判定の大きさを超えてしまい、結果として当たり判定をすり抜けてしまう事があります。

図8－7－1 高速で移動すると、当たり判定が抜けるケースのイメージ図

当たり判定
領域
次のフレーム

高速で移動したり当たり判定自体が小さかったりすると接触せずにすり抜けてしまう

すり抜けを解決するには

解決方法として、移動速度を落としたり当たり判定を大きめに取るなどの手法が考えられます。
しかし、ゲームバランス上、上記の解決方法が難しい場合は、別の手法を用いて解決しなくてはなりません。
他の解決法には幾つかありますが、比較的手軽な方法として、当たり判定のチェック回数を増やしてやる方法があります。

要するに、1フレームの移動で判定をすり抜けてしまうのであれば、その中間の当たり判定を仮想的に作り出して、その当たり判定に対してもチェックを行なってやるのです。
こうする事で、当たり判定のすり抜けを防ぐ事が出来ます。

すり抜け防止のプログラム

では、実際のプログラムを見ていきます。サンプルは、方向キーでボールの移動を行ない、画面のボールとの当たり判定のチェックを行なうものです。

通常時はすり抜けてしまいますが、Zキーを押しながら移動させると当たり判定が切り替わり、接触のメッセージが表示されるようになります。

メイン処理

まず、メインのプログラムです。キー入力による移動処理後、当たり判定のための矩形を計算しています。

その後、当たり判定のチェックを行ないます。ここは、Zキーの入力で2種類の判定関数を切り替えています。

チェック後、現在の判定の矩形をを、1フレーム前の当たり判定として記録しておきます。

後は表示を行ない、メインの処理は終了です。

当たり判定チェック関数

次に、当たり判定チェック用の関数について解説します。

まず、EX08_07_hit_check ですが、これは通常の矩形と点の当たり判定を行ないます。

こちらの詳細に関しては、[8－3]を参照してください。

次に、EX08_07_hit_check_double ですが、こちらは現在の当たり判定矩形と1フレーム前の矩形を元に当たり判定を行ないます。

最初に、1フレーム前の当たり判定矩形と現在の矩形から、仮想の当たり判定矩形を算出します。

これは、2つの矩形の中間点を取得する事と同義ですので、矩形の各座標を加算して、2で割ってやる事で取得できます。

そして、現在の当たり判定と、今作成した当たり判定の2つの当たり判定に対してチェックを行なえば、当たり判定は完成です。

注意点

最後に注意点ですが、当たり判定が非常に小さい場合は、このチェック関数でも、まだすり抜けるケースがあります。

その場合は、作成する矩形を更に増やしてやると良いでしょう。ただし、その分処理が重くなってしまうので扱いには注意してください。

LIST 8 —7—1 スピードアップ時の当たり判定

```
typedef struct{
    SPRITE          Ball;
    SPRITE          Target;
    RECT            OldHitRect;
} EX08_07_STRUCT;

int EX08_07_hit_check_double( RECT NowHitRect, RECT BeforeRect, float
checkX, float checkY )
{
    RECT HalfHitRect;

    //判定チェックのための矩形を計算
    HalfHitRect.top    =   (NowHitRect.top    + BeforeRect.top    ) / 2;
    HalfHitRect.bottom =   (NowHitRect.bottom+ BeforeRect.bottom) / 2;
    HalfHitRect.left   =   (NowHitRect.left   + BeforeRect.left   ) / 2;
    HalfHitRect.right  =   (NowHitRect.right  + BeforeRect.right  ) / 2;

    //対象と2つの当たり判定の矩形を比較
      //1つ目の矩形(1つ前の座標との中間位置)
    if((HalfHitRect.top    < checkY  &&
        HalfHitRect.bottom> checkY  &&
        HalfHitRect.left   < checkX  &&
        HalfHitRect.right  > checkX  )||
      //2つ目の矩形(現在の当たり判定)
       (NowHitRect.top     < checkY  &&
        NowHitRect.bottom  > checkY  &&
        NowHitRect.left    < checkX  &&
        NowHitRect.right   > checkX  ))
    {
        return true;
    }

    return false;
}

int EX08_07_hit_check( RECT NowHitRect, float checkX, float checkY )
{
    //対象と当たり判定の矩形を比較
    if( NowHitRect.top     < checkY  &&
```

```
        NowHitRect.bottom > checkY  &&
        NowHitRect.left   < checkX  &&
        NowHitRect.right  > checkX  )
    {
        return true;
    }

    return false;
}

void exec08_07(TCB* thisTCB)
{
#define MOVE_SPEED  32.0
#define TARGET_CENTER  16.0

    EX08_07_STRUCT* work = (EX08_07_STRUCT*)thisTCB->Work;
    int  hit_flag = 0;
    RECT hit_size_rect = { 12,12, 20, 20, };
    RECT hit_rect;
    char str[128];
    RECT font_pos = { 0, 0, 640, 480, };

    //キー入力による移動
    if( g_DownInputBuff & KEY_UP    ) work->Ball.Y -= MOVE_SPEED;
    if( g_DownInputBuff & KEY_DOWN  ) work->Ball.Y += MOVE_SPEED;
    if( g_DownInputBuff & KEY_RIGHT ) work->Ball.X += MOVE_SPEED;
    if( g_DownInputBuff & KEY_LEFT  ) work->Ball.X -= MOVE_SPEED;

    //判定チェックのための矩形を計算
    hit_rect.top    =   work->Ball.Y + hit_size_rect.top;
    hit_rect.bottom =   work->Ball.Y + hit_size_rect.bottom;
    hit_rect.left   =   work->Ball.X + hit_size_rect.left;
    hit_rect.right  =   work->Ball.X + hit_size_rect.right;


    //Zキーを押しながらだと当たり判定を変える
    if( g_InputBuff & KEY_Z )
    {//対象が現在と過去の矩形の範囲内にあれば当たっている
        hit_flag = EX08_07_hit_check_double( hit_rect,  work->OldHitRect,
                                    work->Target.X+TARGET_CENTER,  //中心位置
```

```
も加算
                                    work->Target.Y+TARGET_CENTER);
    }else{
     //現在の当たり矩形のみをチェック
        hit_flag = EX08_07_hit_check( hit_rect,
                              work->Target.X+TARGET_CENTER,            //目標の中
                                                                       心位置も加算
                              work->Target.Y+TARGET_CENTER);
    }
    //接触時のチェック
    if(hit_flag)
    {
        sprintf( str,"接触しました");
        g_pFont->DrawText( NULL, str, -1, &font_pos, DT_LEFT, 0xffffffff);
    }

    //現在の当たり矩形を保存
    work->OldHitRect = hit_rect;

    SpriteDraw(&work->Ball,0);
    SpriteDraw(&work->Target,0);
}
```

8-8 自機の当たり判定の調整

自機と当たり判定

ここでは、シューティングの自機と当たり判定について、少し触れてみます。

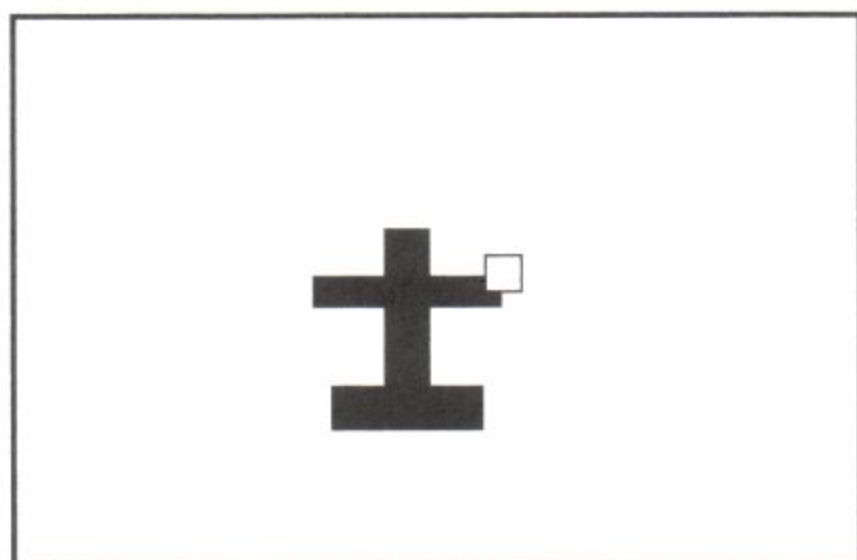
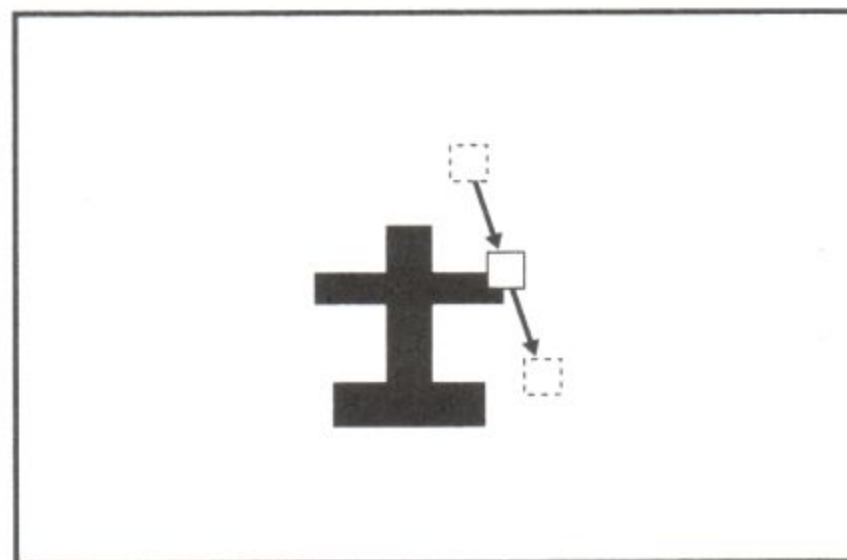
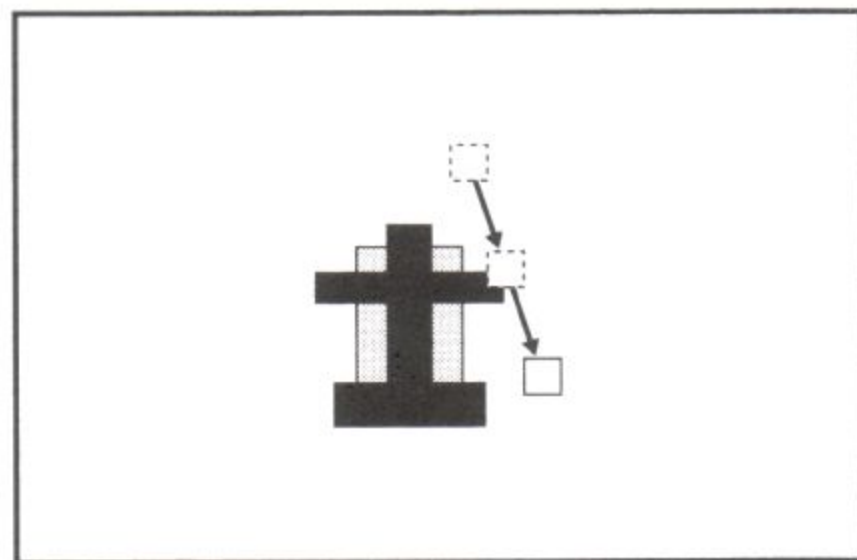
基本的に当たり判定は、物体と物体が接触した時に起こる物で、ゲームではスプライトがそれにあたります。

ただ、スプライトには透明な領域があるため、実際にはスプライトに描かれている絵の大きさの通りに当たり判定をあわせるのが正しい、と言う事になります。

しかしこの場合、理論的には正しくても、プレイヤー的は納得しない事があります。

なぜなら、余りにも正確に当たり判定を取ると、わずか1ドット重なっただけで、当たった事になってしまい「当たり判定の厳しいゲーム」と感じられてしまうからです。

実際当たっているのですから、しょうがないといえばそうなのですが、ゲーム上では常に自機や弾が動いているためか、ほんの一瞬一部が重なっただけだと、当たったとは認識されにくいようです。

図8－8－1 正確な当たり判定だと、認識しにくいため、プレイヤーは納得しにくい図

自機にわずか1ドットの弾が当たっただけで死んでしまう

しかも、実際に当たったのはほんの一瞬の為、納得がいかない

当たり判定は実際の大きさより小さめの方が納得しやすい

そのため、対象のゲームにもよりますが、通常は余り正確に当たり判定をとらず、見た目より少し小さな当たり判定を持ったほうが、プレイヤーは納得するようです。

弾幕系のシューティング

ちなみに、ここで通常、と書いたのは、最近「弾幕系」と呼ばれる画面上を埋め尽くす程、大量の弾を発射するゲームが出てきているからです。

この「弾幕系」と呼ばれるゲームは、大量の弾をかわす事がゲームの基本となります。

そのため、一見「当たっているように見えても当たっていない」程小さな当たり判定が好まれます。

どう見ても当たっているように見えても、プレイヤーにとっては「かすっただけ」とみなされるのか、命中した事にはならず、納得してもらえるようです。

また、当たり判定が小さいため、[8－7]で紹介したような、弾のすり抜けも結構な確率で起ります。

しかしこれも「かわした」とみなされるのか、余り問題にはならず、さほど気にしないで良いのが実情のようです。

8-9 キャラクター同士の詳細な当たり判定

単純な形を組み合わせて判定

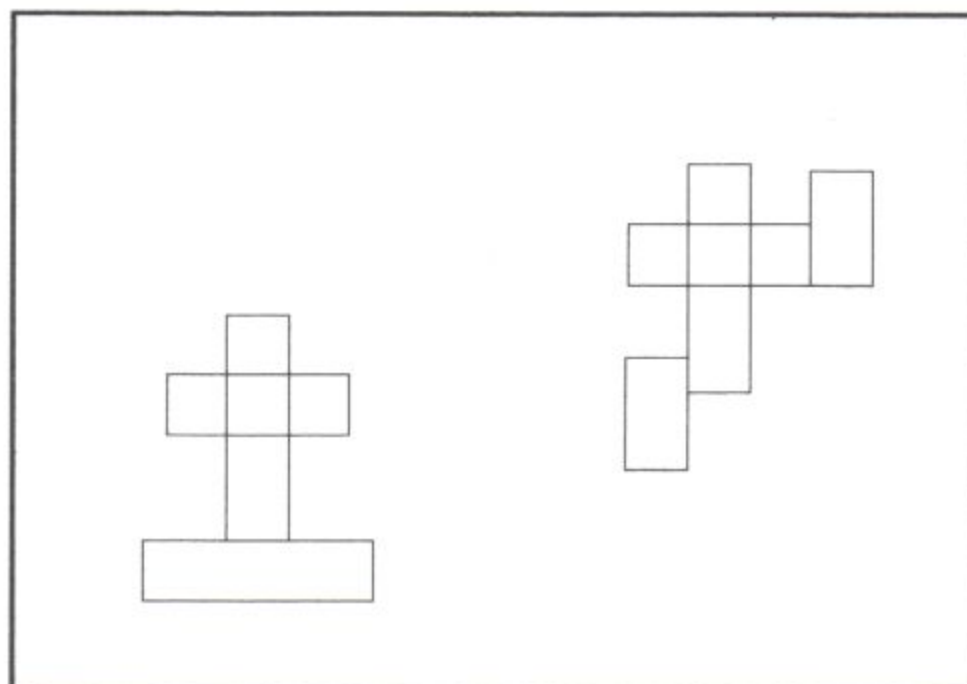
今までに挙げた手法では、円や矩形等、単純な形状しか判定できませんでした。

しかし、実際にゲームを作成していくと、このような単純な判定だと違和感が出てきてしまい、ゲーム性を損ねてしまう場合があります。

そこでここでは、キャラクター同士の詳細な当たり判定について考えてみます。

とはいったものの複雑な形状の当たりを、直接判定する事は簡単ではありません。

そのためここでは、少し発想を変えて、複雑な形状を直接判定するのではなく、単純な当たり判定を組み合わせて、より複雑な当たり判定を構築し、それに対して判定を行なう方法を紹介します。

図8－9－1 単純な形状を用いて当たり判定を構築するイメージ図

単純な形状（ここでは矩形）を複数組み合わせて
複雑な形状を構築する

複雑な当たり判定のプログラム

判定データの用意

では、早速プログラムを見ていきます。サンプルは、凹凸のついた物体同士の当たりを表示するもので、判定は矩形同士の当たり判定を複数組み合わせる事で実現しています。

まず、はじめに当たり判定のデータを用意します。

判定データは矩形の配列で用意し、最後の配列は終了を表すために全要素に－1を入れる様にします。ここでは、3つの矩形を定義しています。

メイン処理

次にメインのプログラムです。まず、キー入力による物体の移動を行ない、その後、判定チェックのための矩形の計算を関数EX08_09_calc_rectで行なっています。

関数の処理内容ですが、元になる当たり判定の配列に、物体の座標を加算します。それを配列の要素だけ繰り返し、最終的に物体の当たり判定の配列を出力する様になっています。

その後、計算した2つの判定配列が、お互いに接触しているか、関数EX08_09_hit_checkでチェックします。

この関数は内部で、2つの配列に格納された矩形を総当りでチェックするようになっており、お互いの矩形の、どれか1つでも接触していれば当たったとみなします。

最後に、総当りという点で気がつく方もいると思いますが、この処理は比較的重い処理です。サンプルでは配列の要素数は3つですが、それでも3×3で、最大9回の比較を行なう可能性があります。

もしこれが10個の配列同士であるならば、最高100回の比較を行なう事になりますので、扱いには注意が必要です。

LIST 8－9－1 キャラクター同士の詳細な当たり判定

```
typedef struct{
    SPRITE         Sprt;
    SPRITE         Target;
} EX08_09_STRUCT;

#define END_CODE -1
int EX08_09_hit_check( RECT* HitRect, RECT* TargetRect)
{
    int hit_loop = 0;
    int target_loop;

    //2つの当たり矩形同士を1つづつ総当りで比較する
    while( HitRect[hit_loop].top != END_CODE )
    {
        target_loop = 0;
        while( TargetRect[target_loop].top != END_CODE )
        {//対象枠全てに対して当たりをチェック、どれか1つでも当たっていれば終了
            if( HitRect[hit_loop].top    <= TargetRect[target_loop].bottom
&&
                HitRect[hit_loop].bottom >= TargetRect[target_loop].top
```

```
&&
                HitRect[hit_loop].left   <= TargetRect[target_loop].right
&&
                HitRect[hit_loop].right  >= TargetRect[target_loop].left    )
            {
                return true;
            }
            target_loop++;
        }
        hit_loop++;
    }
    return false;
}

void EX08_09_calc_rect( RECT* AnsRect, RECT* BaseHitRect,float AddX,float
AddY)
{
    int loop = 0;

    while( BaseHitRect[loop].top != END_CODE )
    {//元座標をから、当たり判定用の矩形を算出
        AnsRect[loop].top    = BaseHitRect[loop].top     + AddY;
        AnsRect[loop].bottom = BaseHitRect[loop].bottom  + AddY;
        AnsRect[loop].left   = BaseHitRect[loop].left    + AddX;
        AnsRect[loop].right  = BaseHitRect[loop].right   + AddX;
        loop++;
    }

    //判定の配列に終了コードを書き込む
    AnsRect[loop].top = -1;
}

void init08_09(TCB* thisTCB)
{
    EX08_09_STRUCT* work = (EX08_09_STRUCT*)thisTCB->Work;

    //使用するテクスチャの読み込み
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥EX07_09_SPRT.png",&g_pTex[0] );
```

```
	//目標の初期座標
	work->Target.X = SCREEN_WIDTH  / 2;
	work->Target.Y = SCREEN_HEIGHT / 2;
}

void exec08_09(TCB* thisTCB)
{
#define MOVE_SPEED  4.0
#define RECT_COUNT  8

	EX08_09_STRUCT* work = (EX08_09_STRUCT*)thisTCB->Work;
	int  hit_flag = 0;
	RECT hit_size_rect[] =
{//当たり判定のデータ
{ 24,  0, 39, 63, },    //1つ目の当たり矩形
{  0, 16, 63, 23, },    //2つ目の当たり矩形
{  0, 56, 63, 63, },    //3つ目の当たり矩形
{ -1, -1, -1, -1, },    //4つ目(データは終了コード)
};
	RECT hit_rect[ RECT_COUNT ];
	RECT target_rect[ RECT_COUNT ];
	char str[128];
	RECT font_pos = { 0, 0, 640, 480, };

	//キー入力による移動
	if( g_InputBuff & KEY_UP    ) work->Sprt.Y -= MOVE_SPEED;
	if( g_InputBuff & KEY_DOWN  ) work->Sprt.Y += MOVE_SPEED;
	if( g_InputBuff & KEY_RIGHT ) work->Sprt.X += MOVE_SPEED;
	if( g_InputBuff & KEY_LEFT  ) work->Sprt.X -= MOVE_SPEED;

	//判定チェックのための矩形を計算
	EX08_09_calc_rect(
		hit_rect, hit_size_rect,
		work->Sprt.X, work->Sprt.Y);

	//同様に目標の判定チェック矩形を計算
	EX08_09_calc_rect(
		target_rect, hit_size_rect,
		work->Target.X, work->Target.Y);
```

```
    //双方の当たり矩形リストをチェック
    if( EX08_09_hit_check( hit_rect, target_rect ) )
    {
        sprintf( str,"接触しました");
        g_pFont->DrawText( NULL, str, -1, &font_pos, DT_LEFT, 0xffffffff);
    }

    SpriteDraw(&work->Sprt,0);
    SpriteDraw(&work->Target,0);
}
```

8-10 一定時間無敵モード

無敵モードの処理

一定時間無敵モードに移行する処理を作成してみましょう。

この処理は、シューティングなどでよく見られる処理です。

ミスにより自機がダメージを受けたり、死亡した際に復活する時に、当たり判定がなくなる処理で、その期間は分かりやすいように自機が点滅していたりします。

処理の手順自体は非常に単純で、指定された時間の間だけ、当たり判定の処理を行なわないようにするだけです。

ただ、このままでは処理がシンプルすぎるので、同時にタスクを利用した複数の物体との当たり判定も実装しています。

一時無敵化の処理

初期化とメイン処理

では、プログラムを見ていきましょう。サンプルは、上から降ってくるボールが自機に接触した際に、一定時間無敵になるものです。

まずは初期化です。グラフィックデータを読み込んだ後、ボールの作成を行ないます。

ボールは関数 exec08_10_make_ball で作成され、タスクとして実行されます。

次にメインの処理です。自機の移動後、自機の当たり判定を行なうための矩形領域を計算します。

当たり判定チェック

その後、当たり判定をチェックするのですが、この際に、無敵モードの時間中かどうかをチェックします。もし時間中であれば当たり判定そのものを行なわないようにします。

もし無敵モードでなければ、当たり判定を行ないます。判定は関数 EX08_10_hit_check 内で行なわれ、もしボールと接触していたら、無敵時間を設定し無敵モードに移行します。

その関数の処理ですが、ここではタスクを使用した自機とボールとの当たり判定を行なっています。

まず、初期化として、関数 GetTaskTop を使用し、タスクリストの先頭アドレスを取得します。その際に一番はじめのタスク（タスクヘッド）は処理の対象からはずすため、飛ばしてやります。

次に、処理の優先順位を利用して、ボールのタスクを識別します。ここでは BALL_PRIORITY

がその値で、もしこの優先順位であればボールの処理タスクとみなし、座標を取得して当たり判定をチェックします。

これを全部のタスクに対して行なえば、全てのボールに対しての当たり判定が行なえます。

表示処理

当たり判定の終了後、メインの最後では表示処理を行ないます。

ここでは、同時に点滅処理も行なっており、もし無敵モード中であれば、タイマーの最下位ビットをみて表示／非表示を切り替えてやります。

LIST 8 －10－1 一定時間無敵

```
#define MOVE_SPEED  8.0
#define BALL_CENTER 16
#define BALL_COUNT 20
#define BALL_SPEED 4
#define BALL_SPEED_MAX 8
#define BALL_PRIORITY 0x2000
#define INVINCIBLE_TIME 50

void exec08_10_ball(TCB* thisTCB);

typedef struct{
    SPRITE          Ball;
    float           AddX;
    float           AddY;
    int             Time;
} EX08_10_BALL;

typedef struct{
    SPRITE          MyShip;
    int             InvincibleTime;
} EX08_10_STRUCT ;


void exec08_10_make_ball( void )
{
    TCB*    ball_tcb;
    EX08_10_BALL* ball_work;

    //ボールの初期化と、座標を設定
```

```
    ball_tcb = TaskMake( exec08_10_ball , BALL_PRIORITY );
    ball_work = (EX08_10_BALL*)ball_tcb->Work;
    ball_work->Ball.X = rand() % SCREEN_WIDTH;
    ball_work->Ball.Y = 0;
    ball_work->AddY   = BALL_SPEED + rand() % BALL_SPEED_MAX;
}

void exec08_10_ball(TCB* thisTCB)
{
    EX08_10_BALL* work = (EX08_10_BALL*)thisTCB->Work;

    //画面上部から移動
    if( work->Ball.Y > SCREEN_HEIGHT )
    {
        //処理終了時にボールを作成する
        exec08_10_make_ball( );
        TaskKill(thisTCB);
        return;
    }

    //ボールの移動処理と表示
    work->Ball.X += work->AddX;
    work->Ball.Y += work->AddY;
    SpriteDraw( &work->Ball,1);
}


int EX08_10_hit_check( RECT* HitRect )
{
    TCB*     check_tcb;
    EX08_10_BALL* ball_work;

    //タスクリストを辿り、ボールのタスクを全部チェックする
    //タスクの先頭アドレスを取得
    check_tcb = GetTaskTop();
    //タスクヘッドは処理せず飛ばす
    check_tcb = check_tcb->Next;
    //タスクヘッドにたどり着いたら1順
    while( check_tcb->Prio != 0x0000)
    {
```

```
        if( check_tcb->Prio == BALL_PRIORITY )
        {//タスクがボールであれば、当たり判定をチェックする
            ball_work = (EX08_10_BALL*)check_tcb->Work;
            //対象と当たり判定の矩形を比較
            if( HitRect->top    < ball_work->Ball.Y + BALL_CENTER &&
                HitRect->bottom > ball_work->Ball.Y + BALL_CENTER &&
                HitRect->left   < ball_work->Ball.X + BALL_CENTER &&
                HitRect->right  > ball_work->Ball.X + BALL_CENTER )
            {
                return true;
            }
        }

        //次にチェックするタスク
        check_tcb = check_tcb->Next;
    }

    return false;
}

void init08_10(TCB* thisTCB)
{
    int loop;

    //使用するテクスチャの読み込み
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥0044.png",&g_pTex[0] );
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥0055.png",&g_pTex[1] );

    //ボールの作成
    for( loop = 0; loop < BALL_COUNT; loop++)
    {
        exec08_10_make_ball();
    }
}

void exec08_10(TCB* thisTCB)
{
    EX08_10_STRUCT* work = (EX08_10_STRUCT*)thisTCB->Work;
```

```
    RECT hit_rect = { 0, 0, 64, 64 };
    char str[128];
    RECT font_pos = { 0, 0, 640, 480, };


    //キー入力による移動
    if( g_InputBuff & KEY_UP    ) work->MyShip.Y -= MOVE_SPEED;
    if( g_InputBuff & KEY_DOWN  ) work->MyShip.Y += MOVE_SPEED;
    if( g_InputBuff & KEY_RIGHT ) work->MyShip.X += MOVE_SPEED;
    if( g_InputBuff & KEY_LEFT  ) work->MyShip.X -= MOVE_SPEED;

    //判定チェックのための矩形を計算
    hit_rect.top    +=   work->MyShip.Y;
    hit_rect.bottom +=   work->MyShip.Y;
    hit_rect.left   +=   work->MyShip.X;
    hit_rect.right  +=   work->MyShip.X;

    if( work->InvincibleTime )
    {//無敵時間中は、当たり判定を行なわない
        //無敵時間を減少させる
        work->InvincibleTime--;
        g_pFont->DrawText( NULL,"無敵時間中!!" , -1, &font_pos, DT_LEFT,
0xffffffff);
    }else{
     //当たり判定をチェックし当たっていたら無敵時間を設定
        if( EX08_10_hit_check( &hit_rect ))
        {
            work->InvincibleTime = INVINCIBLE_TIME;
        }
    }

    //点滅処理と表示処理
    if(!(work->InvincibleTime & 0x01)) SpriteDraw(&work->MyShip,0);
}
```

8-11 広範囲ミサイル

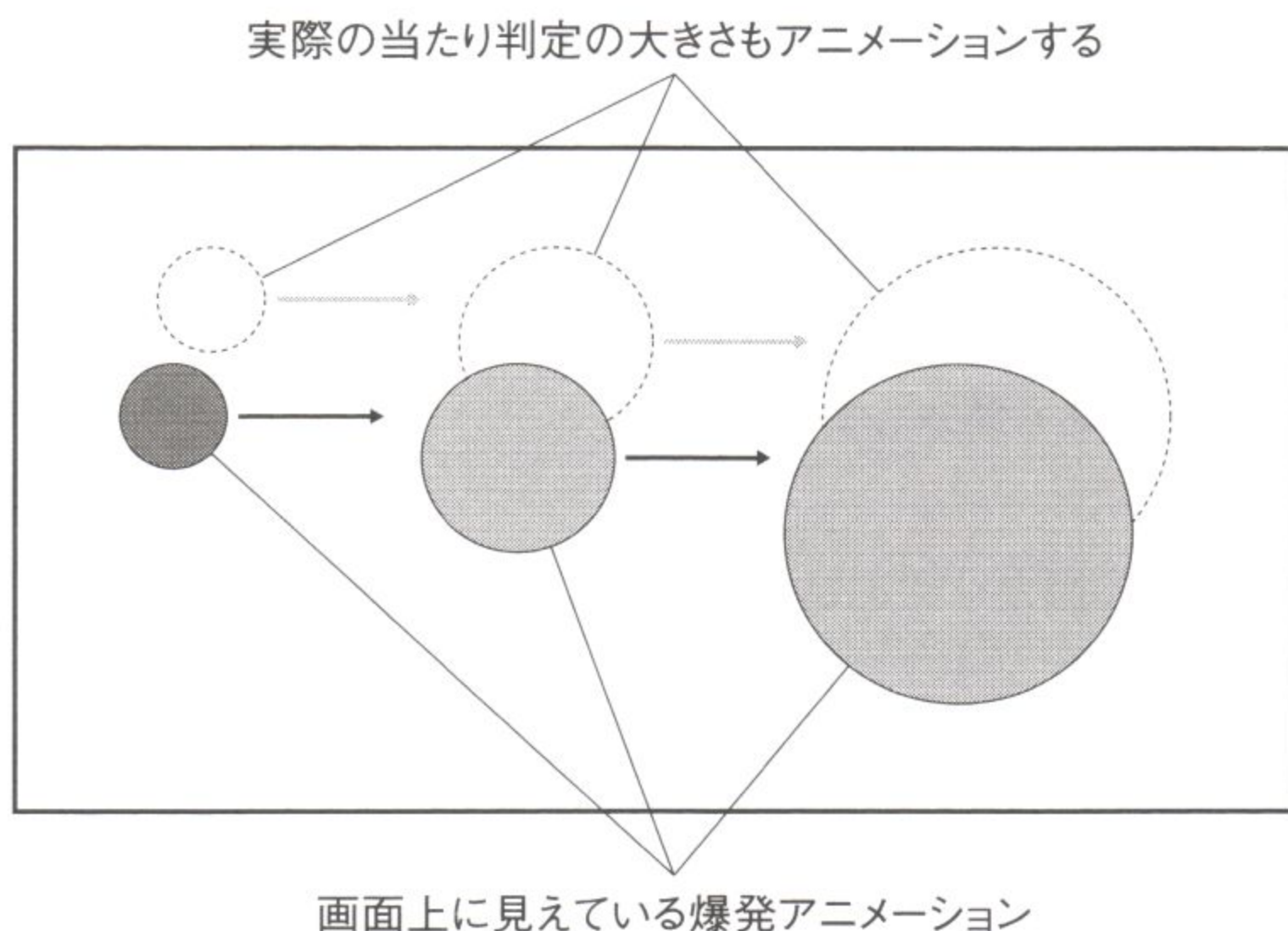
爆発や爆風が広がる処理

広範囲に影響を与える、爆発の処理を考えてみましょう。

着弾後に爆発や爆風が広がる処理は、通常弾、いわゆるショットとは違いミサイルやボムとして知られています。

この処理がショットと違う所は、大雑把に分けて２点あります。

１つは爆発を行なうため、当たり判定が時間によって変化する所。もう１つは、同時に爆発のアニメーションを行なう所です。

図8 －11 －1 爆発時の当たりアニメーションのイメージ図

爆発や爆風が広がるプログラム

概要

ここでは、この２つの爆発の処理と、爆発に当たったボールが消える処理を実装してみましょう。

サンプルプログラムですが、Ｚキーを押す事で画面中央で爆発のアニメーションが起こります。

この時当たり判定も変化しており、それに接触したボールが消える様になっています。

初期化とメイン処理

まず、はじめに初期化として、ボールの作成を行ないます。

ここでは、関数の再利用例として、[8－10]で使用した関数exec08_10_make_ballを使用してボールの作成を行なっています。

次に、メインの処理です。

はじめにZキーで爆発処理（＝アニメーションの開始）のチェックを行ないます。

ただ、ここで行なうのはアニメーションの開始処理として、アニメーションのスピードを設定してやるだけです。

爆発のアニメーション

次に実際のアニメーション処理です。

まず、現在表示されるアニメーションのコマ数、AnimCountから、アニメーション表示用のグラフィック座標を取得し、爆発の表示を行ないます。

グラフィックの座標データは配列で格納されており、要素数は爆発のコマ数と合わせて4つです。

その後爆発とボールの当たり判定を行ないますが、その際、当たり判定のデータも表示されるコマに合わせて取得され、データ配列の要素数も同じく4つになります。

当たり判定の処理は関数EX08_11_hit_checkで行なっており、この関数は当たり判定として、爆発の中心座標と、そこからの判定距離を要求します。

関数の内部処理ですが、[8－10]の当たり判定処理と同様に、タスクを辿って、ボールのタスクを検索します。

そして、爆発範囲とボールとの判定を行なうのですが、この際に直接ボールを消すのではなく、ボールの表示座標を画面外に移動させる事によって消去処理を行なっています。

もちろん直接消しても構わないのですが、消去後、ボールの作成処理も加わるため、こちらの方が記述が簡潔で済みます。

最後に、爆発の処理が終了したら、アニメーションの更新処理を行ないます。

これは、アニメーション速度を、アニメーションのカウンタに加算してあげるだけですので、特に難しくはないでしょう。

LIST 8 ー11ー1 広範囲ミサイル

```
#define BOMB_CENTER 32
#define ANIM_SPEED 0.2
#define ANIM_COUNT 4
#define BALL_COUNT 20

typedef struct{
    SPRITE          Bomb;
    float           AnimCount;
    float           AnimSpeed;
} EX08_11_STRUCT ;

int EX08_11_hit_check( float PosX, float PosY,float HitRange )
{
    TCB*    check_tcb;
    EX08_10_BALL* ball_work;
    float distance;
    float ball_X;
    float ball_Y;
    float pos_X;
    float pos_Y;

    //タスクリストを辿り、ボールのタスクを全部チェックする
    check_tcb = GetTaskTop();               //タスクの先頭アドレスを取得
    check_tcb = check_tcb->Next;            //タスクヘッドは処理せず飛ばす
    while( check_tcb->Prio != 0x0000)       //タスクヘッドにたどり着いたら1順
    {
        if( check_tcb->Prio == BALL_PRIORITY )
        {//タスクがボールであれば、当たり判定をチェックする
            ball_work = (EX08_10_BALL*)check_tcb->Work;
            //座標を物体の中心点に合わせる
            ball_X = ball_work->Ball.X + BALL_CENTER;
            ball_Y = ball_work->Ball.Y + BALL_CENTER;
            pos_X  = PosX + BOMB_CENTER;
            pos_Y  = PosY + BOMB_CENTER;

            //同一座標の場合エラーが出るので特別処理にする
            if( ball_X == pos_X && ball_Y == pos_Y )
            {//座標を書き換え強制的に画面外へ移動させ、消去
                ball_work->Ball.Y = SCREEN_HEIGHT;
```

```
                }else{
                    //相手との距離を計測し、当たり範囲内なら画面外へ移動させ、消去
                    distance = sqrtf( (ball_X - pos_X) * (ball_X - pos_X) +
                                    (ball_Y - pos_Y) * (ball_Y - pos_Y) );
                    if( distance <= HitRange )  ball_work->Ball.Y =
SCREEN_HEIGHT;
                }
            }
            //次にチェックするタスク
            check_tcb = check_tcb->Next;
        }

        return false;
}

void init08_11(TCB* thisTCB)
{
    EX08_11_STRUCT* work = (EX08_11_STRUCT*)thisTCB->Work;
    int loop;

    //使用するテクスチャの読み込み
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥spread.png",&g_pTex[0] );
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥0055.png",&g_pTex[1] );

    //ボールの作成
    for( loop = 0; loop < BALL_COUNT; loop++)
    {//前項目の関数を再利用
        exec08_10_make_ball();
    }
    work->Bomb.X = SCREEN_WIDTH / 2;
    work->Bomb.Y = SCREEN_HEIGHT / 2;
}

void exec08_11(TCB* thisTCB)
{
    EX08_11_STRUCT* work = (EX08_11_STRUCT*)thisTCB->Work;
    float hit_range[ ANIM_COUNT ] = { 16.0, 32.0, 48.0, 64.0,};    //当たり判定
の大きさ
```

```
    RECT anim_table[ ANIM_COUNT] =
{
    {  0,  0, 64, 64,},    //1コマ目
    { 64,  0,128, 64,},    //2コマ目
    {128,  0,192, 64,},    //3コマ目
    {192,  0,256, 64,},    //4コマ目
};


    //Zキーで爆発処理開始
    if( g_DownInputBuff & KEY_Z  )
    {//ここでは、アニメーションの開始を行なうだけ
        work->AnimSpeed = ANIM_SPEED;
    }

    //アニメーションが始まっていれば表示
    if(work->AnimSpeed != 0.0)
    {
        //現在表示のフレームからアニメーション用の座標を取得
        work->Bomb.SrcRect = &anim_table[(int)work->AnimCount];

        //表示
        SpriteDraw(&work->Bomb,0);

        //爆発の処理、座標と範囲を渡す
        EX08_11_hit_check( SCREEN_WIDTH / 2, SCREEN_HEIGHT / 2,
hit_range[(int)work->AnimCount]);

        //アニメーションを進める
        work->AnimCount += work->AnimSpeed;
        //指定のコマ数に達していればアニメーション終了
        if(work->AnimCount >= ANIM_COUNT)
        {
            work->AnimCount = 0;
            work->AnimSpeed = 0;
        }
    }
}
```

8-12 まとめて敵を吹き飛ばす

回避用の吹き飛ばし技

アクションゲームでは、ボタンの同時押し等で、周りの敵を吹き飛ばす処理が良く見られます。
ここでは、この処理を実装してみましょう。

処理の概要

処理の概要ですが、大きく3つの部分に分けられます。
1つは、メインとなる移動、入力処理、もう1つは当たり判定による吹き飛ばし対象の処理の切り替え、そして最後の1つは対象の処理そのものです。

吹き飛ばし技のプログラム

では、実際の処理を見ていきましょう。サンプルは方向キーで自機を動かし、Zキーで自機の周りにあるボールを吹き飛ばすものです。

さて、最初にプログラムを読まれた方は気づいたかと思いますが、実はこの処理は、[8－10]と基本的な処理が近く、実際の処理もその延長線上にあります。
また、関数もボールの作成処理等は[8－10]の処理を再利用しており、大きな処理ではありません。
そのためここでは、吹き飛ばしの処理を行なうEX08_12_hit_checkの解説を重点的に行ないます※。

吹き飛ばしの処理

まず、ボールタスクの検索を行ない、吹き飛ばし判定のチェックを行ないます。この際のチェックは[8－1]で解説した、対象との距離を計る事で行ないます。
もし範囲内にボールがあれば、対象のボールタスクをTaskChangeを用いて切り替えます。
切り替え先は、ボール吹き飛ばし処理の関数、EX08_12_bash_ballで、切り替え処理の際は、自機を中心とした反対側の方向へ進むように、速度を設定してやります。

ここで理解して欲しいポイントは、この処理用に作られた訳ではないタスク処理を、TaskChangeで切り替える事で、この処理専用にしてしまう事です。

※この項目だけでわかりにくいと感じた方は、[8－10]の方も参考にしてみてください。

最後は、吹き飛ばされるボールの処理関数、EX08_12_bash_ball です。

速度を設定されたタスクは一定時間進んだ後に、消去されます。そのとき同時に、ボールの作成関数を呼び出します。

こうする事で、画面上のボールの数を常に一定に保つ事が出来ます。

LIST 8 －12－1 まとめて敵を吹き飛ばす

```
#define MOVE_SPEED  8.0
#define MYSHIP_CENTER 32
#define BALL_COUNT 20
#define BALL_BASH_SPEED 24
#define BALL_PRIORITY 0x2000
#define HIT_RANGE 150
#define BALL_BASH_TIME 5

typedef struct{
    SPRITE          MyShip;
} EX08_12_STRUCT ;

void EX08_12_bash_ball( TCB* thisTCB )
{
    EX08_10_BALL* work = (EX08_10_BALL*)thisTCB->Work;

    //一定時間のみ移動
    if( work->Time++ > BALL_BASH_TIME )
    {
        //処理終了時にボールを作成する
        exec08_10_make_ball( );
        TaskKill(thisTCB);
        return;
    }
    //ボールの移動処理と表示
    work->Ball.X += work->AddX;
    work->Ball.Y += work->AddY;
    SpriteDraw( &work->Ball,1);
}

int EX08_12_hit_check( float PosX, float PosY )
{
```

```
TCB*    check_tcb;
EX08_10_BALL* ball_work;
float distance;
float direction;
float ball_X;
float ball_Y;
float pos_X;
float pos_Y;


//タスクリストを辿り、ボールのタスクを全部チェックする
//タスクの先頭アドレスを取得
check_tcb = GetTaskTop();
//タスクヘッドは処理せず飛ばす
check_tcb = check_tcb->Next;
//タスクヘッドにたどり着いたら1順
while( check_tcb->Prio != 0x0000)
{
    if( check_tcb->Prio == BALL_PRIORITY )
    {//タスクがボールであれば、当たり判定をチェックする
        ball_work = (EX08_10_BALL*)check_tcb->Work;
        //座標を物体の中心点に合わせる
        ball_X = ball_work->Ball.X + BALL_CENTER;
        ball_Y = ball_work->Ball.Y + BALL_CENTER;
        pos_X  = PosX + MYSHIP_CENTER;
        pos_Y  = PosY + MYSHIP_CENTER;

        //同一座標の場合エラーが出るので特別処理にする
        if( ball_X == pos_X && ball_Y == pos_Y )
        {
            ball_work->AddX = 0;
            ball_work->AddY = 0;
            TaskChange( check_tcb, EX08_12_bash_ball);
        }else{
            //相手との距離を計測
            distance = sqrtf( (ball_X - pos_X) * (ball_X - pos_X) +
                              (ball_Y - pos_Y) * (ball_Y - pos_Y) );
            if( distance <= HIT_RANGE )
            {//吹き飛ばしの範囲内であればボールの速度を自機とは反対の方向へ設定する
                //座標から相手への方向を計算
```

```
                        direction = atan2( ball_Y - pos_Y, ball_X - pos_X );
                        //方向から、X、Yそれぞれの速度を計算
                        ball_work->AddX = cos( direction ) * BALL_BASH_SPEED;
                        ball_work->AddY = sin( direction ) * BALL_BASH_SPEED;
                        //外部から、ボール処理を吹き飛ばし処理に切り替える
                        TaskChange( check_tcb, EX08_12_bash_ball);
                    }
                }
            }
            //次にチェックするタスク
            check_tcb = check_tcb->Next;
        }

    return false;
}

void init08_12(TCB* thisTCB)
{
    int loop;

    //使用するテクスチャの読み込み
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥0044.png",&g_pTex[0] );
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥0055.png",&g_pTex[1] );

    //ボールの作成
    for( loop = 0; loop < BALL_COUNT; loop++)
    {//前項目の関数を再利用
        exec08_10_make_ball();
    }
}

void exec08_12(TCB* thisTCB)
{
    EX08_12_STRUCT* work = (EX08_12_STRUCT*)thisTCB->Work;

    //キー入力による移動
    if( g_InputBuff & KEY_UP     ) work->MyShip.Y -= MOVE_SPEED;
    if( g_InputBuff & KEY_DOWN   ) work->MyShip.Y += MOVE_SPEED;
```

```
    if( g_InputBuff & KEY_RIGHT ) work->MyShip.X += MOVE_SPEED;
    if( g_InputBuff & KEY_LEFT  ) work->MyShip.X -= MOVE_SPEED;

    //Zキーで吹き飛ばし処理
    if( g_DownInputBuff & KEY_Z  )
    {//当たり判定をチェックし吹き飛ばす処理
        EX08_12_hit_check( work->MyShip.X, work->MyShip.Y );
    }
    //表示
    SpriteDraw(&work->MyShip,0);
}
```

8-13 キャラクター同士の当たり判定（対戦格闘）

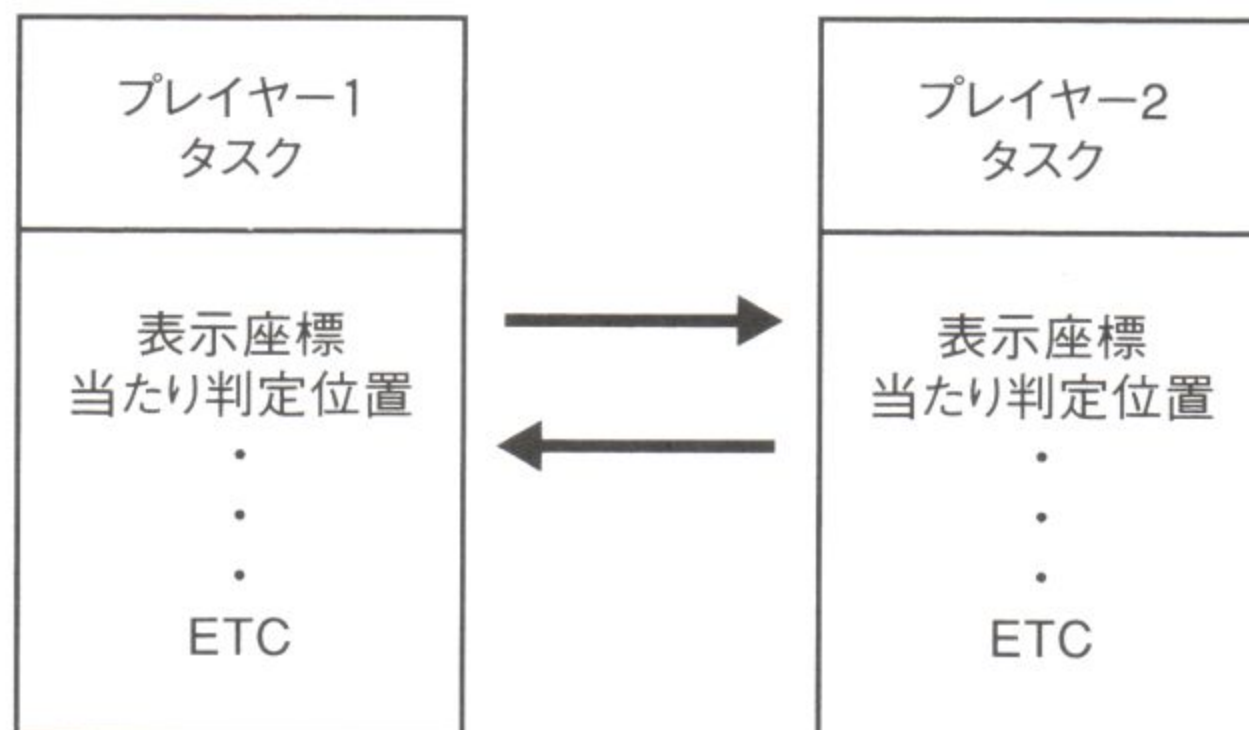
対戦時における情報のやり取り

対戦格闘ゲーム等の２人同時プレイにおける当たり判定を考えて見ましょう。

通常、こういった２人同時プレイなどの処理を行なう場合、２つのタスク(処理）を作成し、お互いに情報をやりとり（通信）しながら処理を行います。

そこでここでは、まず基本となる、相互の当たり判定処理を行なってみます。

図8－13－1 タスクがお互いにやり取りするイメージ図

プレイヤー1 タスク		プレイヤー2 タスク
表示座標 当たり判定位置 ・ ・ ・ ETC	→ ←	表示座標 当たり判定位置 ・ ・ ・ ETC

同時プレイの場合で当たり判定を行なう場合などは
必要な情報をお互いにやり取りする

相互のタスクの作成

解説の前に、この処理は、[7－4]複数同時プレイの処理を基本にしています。読まれてない方は、まずはそちらを参照してください。

では早速プログラムを見ていきます。

まずは初期化処理です。はじめに画像データを読み込みます。これは1P側のプレイヤーと2P側のプレイヤーのデータで、反転を行なっている以外は同じ画像データです。

その後、２つのタスク（プレイヤー処理用のタスク）を作成しています。

処理関数は同じ関数で、唯一、登録するプレイヤーID、PIDが違います。

このプレイヤーIDで1P側のプレイヤーと2P側のプレイヤーを区別します。

作成後、通信処理の為、お互いのタスクへのポインタをそれぞれのタスクワークに格納し、初期化処理は終了します。

相手との当たり判定

次にメインの処理関数exec08_13_humanを解説します。

まずはじめに、相手側プレイヤーのワークを扱いやすいよう、enemy_workに代入します。

そして、1フレーム前の座標を保存して、キー入力処理を行ないます。この時行なう座標の保存は、移動処理が行えなかった時に、座標を戻す為です。

その後、相手と自分との当たり判定を行ないます。

当たり判定は2つの関数から出来ており、1つはXY座標と矩形領域データから、実際の画面の当たり判定矩形領域を計算する関数、もう1つは2つの矩形領域の当たりを比較する関数です。

まず、先の関数、EX08_13_pos_add_rectを用い、自分と相手の当たり判定座標を取得します。

この時お互いの座標と、ベースになる領域が必要になりますので、当たり判定の領域データとしてhuman_hit_rectを引数に渡してやります。

取得後、当たり判定用の関数、EX08_13_hit_rectを用い、お互いの当たり判定を比較します。

比較後、もし接触していたら、相手にぶつかっているとみなせますので、先に保存しておいた座標を用いて、移動をキャンセルします。

あとは表示を行ない、処理は終了です。

LIST 8 －13－1 キャラクター同士の当たり判定

```
typedef struct{
    SPRITE          Human;
    int             PID;
    TCB*            EnemyPlayerTask;
} EX08_13_STRUCT;

#define MOVE_SPEED      4.0
#define PLAYER_1        0
#define PLAYER_2        1

#define HUMAN_WIDTH     64
#define HUMAN_HEIGHT    96
```

```
void EX08_13_pos_add_rect( RECT* ans_rect, RECT* add_rect, float x, float y
)
{//矩形領域に座標を加算
    ans_rect->top    = add_rect->top    + y;
    ans_rect->bottom = add_rect->bottom + y;
    ans_rect->left   = add_rect->left   + x;
    ans_rect->right  = add_rect->right  + x;
}

int EX08_13_hit_rect( RECT* hit_rectA, RECT* hit_rectB )
{//2つの矩形の接触判定
    //対象枠全てが対となる矩形の範囲内にあれば当たっている
    if( hit_rectA->top    < hit_rectB->bottom &&
        hit_rectA->bottom > hit_rectB->top    &&
        hit_rectA->left   < hit_rectB->right  &&
        hit_rectA->right  > hit_rectB->left   )
    {//ヒット!
        return( true );
    }
    //ヒットしていない
    return( false );
}

unsigned char EX08_13_player_input( int PlayerID )
{    unsigned char input_key;

    //入力値をプレイヤーによって変更する
    switch( PlayerID )
    {//キーボードをプレイヤー1、ジョイスティックをプレイヤー2とする
       case PLAYER_1: input_key = g_KeyInputBuff; break;
       case PLAYER_2: input_key = g_JoystickBuff; break;
    }
    return input_key;
}

void exec08_13_human(TCB* thisTCB)
{
    EX08_13_STRUCT* work = (EX08_13_STRUCT*)thisTCB->Work;
    //相手側のプレイヤーのタスク
    EX08_13_STRUCT* enemy_work = (EX08_13_STRUCT*)work->EnemyPlayerTask-
```

```
>Work;
    //1フレーム前の座標
    float old_posX;
    float old_posY;
    unsigned char input_buff;
    RECT anim_rect = {0,0,64,128,};
    RECT human_hit_rect  = {0,0,64,128,};
    RECT my_hit_rect;
    RECT enemy_hit_rect;

    //1フレーム前の座標を保持
    old_posX = work->Human.X;
    old_posY = work->Human.Y;

    //プレイヤーキャラの表示
    work->Human.SrcRect = &anim_rect;

    //プレイヤーIDによる入力
    input_buff = EX08_13_player_input( work->PID );

    //キー入力による移動
    if( input_buff & KEY_RIGHT ) work->Human.X += MOVE_SPEED;
    if( input_buff & KEY_LEFT  ) work->Human.X -= MOVE_SPEED;

    //自キャラの当たり判定を取得
    EX08_13_pos_add_rect(
            &my_hit_rect, &human_hit_rect,
            work->Human.X, work->Human.Y);

    //相手キャラ自の当たり判定を取得
    EX08_13_pos_add_rect(
            &enemy_hit_rect, &human_hit_rect,
            enemy_work->Human.X, enemy_work->Human.Y);

    //当たり判定をチェック
    if( EX08_13_hit_rect( &my_hit_rect, &enemy_hit_rect ) )
    {//もし接触していたら
     //移動できないので座標を戻す
       work->Human.X = old_posX;
       work->Human.Y = old_posY;
```

```
	}

	//自キャラの描画
	SpriteDraw( &work->Human,work->PID);
}

void init08_13(TCB* thisTCB)
{
	TCB*	human_tcb[2];
	EX08_13_STRUCT* human_work[2];
	int loop;
	float player_pos[][2] =
{//プレイヤーの初期座標
{SCREEN_WIDTH / 4.0 * 1.0,SCREEN_HEIGHT / 2.0,}, //PLAYER1
{SCREEN_WIDTH / 4.0 * 3.0,SCREEN_HEIGHT / 2.0,}, //PLAYER2
};

	//使用するテクスチャの読み込み
	D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥human_anim.png",&g_pTex[0] );
	D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥human_anim_r.png",&g_pTex[1] );

	//プレイヤーを2体作成する
	for( loop = 0;loop < 2; loop++ )
	{
		human_tcb[loop] = TaskMake( exec08_13_human, 0x2000 );
		human_work[loop] = (EX08_13_STRUCT*)human_tcb[loop]->Work;
		//プレイヤーにより初期表示座標を変える
		human_work[loop]->Human.X = player_pos[loop][0];
		human_work[loop]->Human.Y = player_pos[loop][1];
		//プレイヤーIDを登録
		human_work[loop]->PID = loop;
	}

	//お互いのタスクワークを保持しておく
	human_work[PLAYER_1]->EnemyPlayerTask
		= human_tcb[PLAYER_2];
	human_work[PLAYER_2]->EnemyPlayerTask
		= human_tcb[PLAYER_1];
```

```
}

void exec08_13(TCB* thisTCB)
{
	//複数同時処理のため、サンプルではメイン処理は何もしない
}
```

8-14 キー入力とアニメーション・当たり判定

攻撃時の当たり判定とアニメーション

前項に続いて対戦格闘時の当たり判定、今度は攻撃時の当たり判定について考えてみます。

基本となる考え方自体は、キャラクター同士の当たり判定と変わらず、お互いの情報を取得しあい、判定を行ないます。

しかしここには、「攻撃時」(攻撃処理)という特別なタイミングと、「攻撃アニメーション」という要素がかかわってきます。

これは単体の処理としてもなかなか面倒な処理です。

そこでここでは、キー入力と、こういった入力後の処理の切り分けについて解説をします。

入力処理とアニメーション

解説の前に、この処理は、[8－13]キャラクター同士の当たり判定を処理の基本としています。初期化等は同じ処理ですので、そちらを参照してください。

初期化後、作成されたタスクのメイン処理ではキャラクター同士の当たり判定と同様、キー入力と移動を行ないます。

この時、同時に攻撃のキー入力判定も行なっています。通常ですとここで、アニメーション処理を行なうのですが、ここでは処理を行わず、TaskChange で処理を切り替えています。

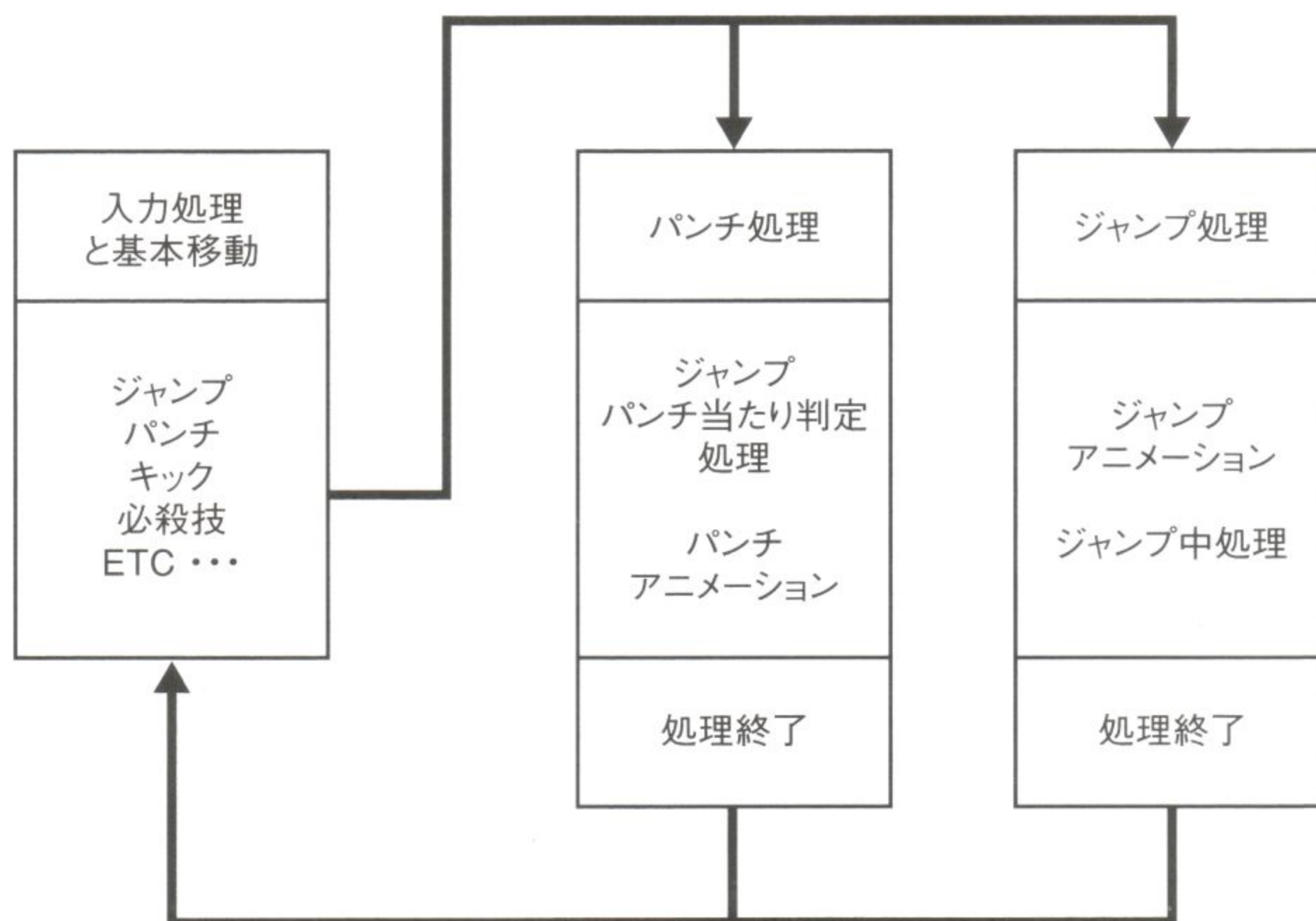
このような事をするのは、先に説明したとおり、入力処理と入力にあわせた各種処理(アニメーション等)の切り分けを行う為です。

特に、サンプルでは攻撃の種類は１種類ですが、実際の処理となるとキックやジャンプ、場合によっては必殺技などの特殊な技もここで処理を行なうことになります。

このように大量の処理が重なる可能性がある所では、プログラムが非常にややこしくなりやすい為、入力側の処理から見ても、こういった処理の切り分けを行なうべきでしょう。

ここでは、一旦他のタスク処理関数に処理を移し、処理先で処理が終了した後に戻ってくるようにしています。

こうする事で、入力処理と他の処理を完全に分離でき、各処理(ここでは攻撃処理と攻撃アニメーション)は処理そのものに専念できます。

図8 －14－1 キー入力と各処理を分離して処理する図

各処理はタスクのため処理が終了するまで各処理に専念できる

8-15 格闘ゲームの攻撃

攻撃時のアニメーション処理

ここでは、実際の攻撃時のアニメーションと攻撃処理について解説していきます。

早速プログラムを見ていきましょう。処理関数は exec08_15_human_attack です。

まずはアニメーションの再生です※。

再生されるコマは変数 AnimFrame から決定され、再生速度は定数 ATTACK_SPEED で決まります。

この時フレーム数をチェックしておき、フレームが最後まで到達、すなわちアニメーション再生が終了したら、処理を元の移動処理 exec08_15_human_move に戻します。

攻撃処理

次に攻撃処理ですが、こちらの方も、アニメーションの再生フレーム数を利用します。

[7－1]キャラのアニメーションでも説明しましたが、アニメーションは細かな画像が集まって出来ています。

この時当然ですが、パンチ攻撃であれば、パンチの画像が表示されます。

そこで画像をチェックし、当たり判定が存在する(またはして欲しい)画像に対してのみ、当たり判定を発生させてやります。

サンプルではパンチの画像が3コマ目ですので3コマ目のみ、攻撃判定が発生します。

この時、攻撃当たり判定の大きさは、human_attack_rect で定義され、大きさは同じですが、1P 側と 2P 側で位置が違います。

これは対戦時互いに向き合っており、攻撃時の判定位置がお互いに違う為です。

なお、攻撃の判定自体は矩形同士の当たり判定を用いています。ここで使う関数は[8－13]で使用したものと同じ物です。

判定処理後、相手に当たっていればメッセージを表示します。

その後表示処理を行い処理は終了です。

※[7－1]キャラのアニメーションも参照してください。

LIST 8 －15－1 格闘ゲームの攻撃

```
#define MOVE_SPEED        4.0
#define PLAYER_1          0
#define PLAYER_2          1

#define HUMAN_WIDTH       64
#define HUMAN_HEIGHT      96
#define ATTACK_SPEED      0.20
#define ATTACK_ANIM_MAX 4.0
#define ATTACK_ANIM_PTN 2

typedef struct{
    SPRITE          Human;
    int             PID;
    TCB*            EnemyPlayerTask;
    float           AnimFrame;
} EX08_15_STRUCT;

void exec08_15_human_move(TCB*);

//プレイヤーの当たり判定領域
static RECT EX08_15_human_hit_rect  = {0,0,64,128,};

void exec08_15_human_attack(TCB* thisTCB)
{//攻撃時の処理
    EX08_15_STRUCT* work = (EX08_15_STRUCT*)thisTCB->Work;
    //対戦側のプレイヤーのタスク
    EX08_15_STRUCT* enemy_work = (EX08_15_STRUCT*)work->EnemyPlayerTask-
>Work;
    RECT my_attack_rect;
    RECT enemy_hit_rect;
    char str[128];

    //攻撃範囲の座標
    RECT human_attack_rect[]  =
{//攻撃判定座標
 { 64, 32, 96, 164,}, //PLAYER_1
 {-32, 32,  0, 164,}, //PLAYER_2
};
    RECT attack_anim_rect[] =
```

```
{
 {  0,   0, 64,128,},   //1コマ目
 {128,   0,192,128,},   //2コマ目
 {192,   0,256,128,},   //3コマ目(攻撃)
 {128,   0,192,128,},   //4コマ目
};

    //プレイヤーキャラのアニメーション表示
    work->Human.SrcRect = &attack_anim_rect[(int)work->AnimFrame];

    work->AnimFrame += ATTACK_SPEED;
    if(work->AnimFrame >= ATTACK_ANIM_MAX)
    {//攻撃アニメが終了したら処理切り替え
        work->AnimFrame = 0;
        TaskChange(thisTCB,exec08_15_human_move);
        //自キャラの描画
        SpriteDraw( &work->Human,work->PID);
    }

    if((int)work->AnimFrame == ATTACK_ANIM_PTN)
    {//攻撃のパターンなら攻撃判定を行なう
        //自キャラの攻撃判定を取得
        EX08_13_pos_add_rect(
                &my_attack_rect,
                &human_attack_rect[work->PID],
                work->Human.X, work->Human.Y);

        //相手キャラ自の当たり判定を取得
        EX08_13_pos_add_rect(
                &enemy_hit_rect, &EX08_15_human_hit_rect,
                enemy_work->Human.X, enemy_work->Human.Y);

        //当たり判定をチェック
        if( EX08_13_hit_rect( &my_attack_rect, &enemy_hit_rect ) )
        {//もし接触していたら
            //攻撃命中とみなし、文字列表示
            sprintf( str,"ATTACK HIT!! P%d",work->PID+1 );
            FontPrint( 128,128+16*work->PID, str);
        }
    }
```

```
        //自キャラの描画
        SpriteDraw( &work->Human,work->PID);
    }


void exec08_15_human_move(TCB* thisTCB)
{
        EX08_15_STRUCT* work = (EX08_15_STRUCT*)thisTCB->Work;
        //対戦側のプレイヤーのタスク
        EX08_15_STRUCT* enemy_work = (EX08_15_STRUCT*)work->EnemyPlayerTask-
>Work;
        //1フレーム前の座標
        float old_posX;
        float old_posY;
        unsigned char input_buff;
        RECT anim_rect = {0,0,64,128,};
        RECT my_hit_rect;
        RECT enemy_hit_rect;

        //1フレーム前の座標を保持
        old_posX = work->Human.X;
        old_posY = work->Human.Y;

        //プレイヤーキャラの表示
        work->Human.SrcRect = &anim_rect;

        //プレイヤーIDによる入力
        input_buff = EX08_13_player_input( work->PID );

        //キー入力による移動
        if( input_buff & KEY_RIGHT ) work->Human.X += MOVE_SPEED;
        if( input_buff & KEY_LEFT  ) work->Human.X -= MOVE_SPEED;

        //キー入力による攻撃
        if( input_buff & KEY_Z )
        {//パンチによる攻撃
            TaskChange(thisTCB,exec08_15_human_attack);
            //自キャラの描画
            SpriteDraw( &work->Human,work->PID);
```

```
        return;
    }


    //自キャラの当たり判定を取得
    EX08_13_pos_add_rect(
            &my_hit_rect, &EX08_15_human_hit_rect,
            work->Human.X, work->Human.Y);

    //相手キャラ自の当たり判定を取得
    EX08_13_pos_add_rect(
            &enemy_hit_rect, &EX08_15_human_hit_rect,
            enemy_work->Human.X, enemy_work->Human.Y);

    //当たり判定をチェック
    if( EX08_13_hit_rect( &my_hit_rect, &enemy_hit_rect ) )
    {//もし接触していたら
     //移動できないので座標を戻す
        work->Human.X = old_posX;
        work->Human.Y = old_posY;
    }

    //自キャラの描画
    SpriteDraw( &work->Human,work->PID);
}
```

Chapter 9

アイテム処理

9-1 アイテム処理の基本

シューティングでのアイテム処理

昨今のゲームにおいて、もはやアイテムの存在しないゲームを探す方が難しいでしょう。

パワーアップをはじめ、スコアアップのためのボーナスや自機の増加、場合によってはイベント進行までアイテムで行なう場合もあります。

ここではリアルタイム、特にシューティングゲームによるアイテム処理について考えてみましょう。

すぐに効果が反映されるアイテム

一般的に、シューティングのアイテムは取得後、すぐに効果が反映されます。

パワーアップやスピードアップ等がそうですし、スコアアップ等のボーナスアイテムもそうでしょう。

このようにアイテムの取得後、即時に効果が反映される様な処理は、当たり判定と、IDによる処理の分岐で実現する事ができます。

すなわち、自機とアイテムの接触判定を行ない、接触したアイテムが保持しているIDに応じて処理を行なってやるのです。

アイテムの作成

それでは実際の処理を見ていきます。

プログラムは、方向キーで自機を左右に動かし、Zキーでランダムに上方から降ってくるボール（アイテム）を取得するものです。

初期化後、まずメインの処理では、キー入力とアイテムの生成処理を行なっています。

アイテムの作成は変数ItemFlagで管理されており、もし画面上のアイテムの出現数等、作成条件を変える時はこの部分を適宜変更すると良いでしょう。

そしてアイテム作成時には、アイテム管理用の構造体EX09_01_ITEMを初期化します。

座標の初期化後、アイテムの種類はランダムで決定しており、種類によってアイテムの色を変更しています。

アイテムの取得と処理

次に、アイテムの取得処理です。まず接触判定ですが、これは当たり判定の章で紹介した、矩形と点の当たり判定を使用しています。

したがってそのサンプルと同様に、判定用の矩形を計算して、自機とアイテムの接触判定を行なっています。

もし、アイテムとの接触があれば、取得したと判断しアイテムに設定したIDを用いて処理を行ないます。

サンプルではパワーアップ等、直接の処理は行なわず、メッセージの変更のみを行なっています。

処理の終了後、ItemFlagをクリアしてアイテムの処理は終了します。

LIST 9 －1－1 アイテム処理の基本

```
#define ITEM_00     0
#define ITEM_01     1
#define ITEM_02     2
#define ITEM_03     3
#define ITEM_MAX    4
#define ITEM_SPEED 5.0
#define MOVE_SPEED 4.0

typedef struct{
        SPRITE          Sprt;
        int             ID;
        float           AddX;
        float           AddY;
        int             Time;
        DWORD           Color;
} EX09_01_ITEM;

typedef struct{
        SPRITE          MyShip;
        int             ItemFlag;
        EX09_01_ITEM    Item;
        char            DispMess[64];
} EX09_01_STRUCT;
```

```
void EX09_01_CustomDraw(SPRITE* pspr,int bitmap_id, DWORD color)
{
    D3DXVECTOR3 pos;

    pos.x = pspr->X;
    pos.y = pspr->Y;
    pos.z = 0;

    g_pSp->Draw( g_pTex[bitmap_id],pspr->SrcRect,NULL,&pos, color);
}

void init09_01(TCB* thisTCB)
{
    EX09_01_STRUCT* work = (EX09_01_STRUCT*)thisTCB->Work;
    //使用するテクスチャの読み込み
    D3DXCreateTextureFromFile( g_pD3Ddevice, "..¥¥..¥¥data¥¥0044.png",&g_pTex[0] );
    D3DXCreateTextureFromFile( g_pD3Ddevice, "..¥¥..¥¥data¥¥0055.png",&g_pTex[1] );

    //自機座標の初期化
    work->MyShip.X = SCREEN_WIDTH  / 2;
    work->MyShip.Y = SCREEN_HEIGHT / 2;
}

void exec09_01(TCB* thisTCB)
{
    EX09_01_STRUCT* work = (EX09_01_STRUCT*)thisTCB->Work;
    RECT hit_size_rect = { 0, 0, 64, 64, };
    RECT hit_rect;
    RECT font_pos = {  0,  0,640,480,};
    DWORD item_color[] =
{//アイテムの色
    0xff0000ff,
    0xff00ff00,
    0xffff0000,
    0xffffffff,
};


    if( g_InputBuff & KEY_RIGHT ) work->MyShip.X += MOVE_SPEED;
    if( g_InputBuff & KEY_LEFT  ) work->MyShip.X -= MOVE_SPEED;
```

```
//Zキーでアイテムを作成
if( g_DownInputBuff & KEY_Z )
{
    //条件によってアイテムは作成しない
    if( !work->ItemFlag )
    {
        //これ以上アイテムが出現しないようにフラグを建てる
        work->Item.Sprt.X = rand() % SCREEN_WIDTH;
        work->Item.Sprt.Y = 0;
        work->Item.AddY    = ITEM_SPEED;
        //アイテムの種類を決定する
        work->Item.ID      = rand() % ITEM_MAX;
        work->Item.Color   = item_color[ work->Item.ID ];
        work->ItemFlag     = true;
    }
}

//アイテム関連の処理

if( work->ItemFlag )
{//アイテム出現時のみ処理
    //アイテム取得時の当たり判定矩形を計算
    hit_rect.top    =   work->MyShip.Y + hit_size_rect.top;
    hit_rect.bottom =   work->MyShip.Y + hit_size_rect.bottom;
    hit_rect.left   =   work->MyShip.X + hit_size_rect.left;
    hit_rect.right  =   work->MyShip.X + hit_size_rect.right;

    //アイテムの取得チェック
    if( hit_rect.top    < work->Item.Sprt.Y  &&
        hit_rect.bottom > work->Item.Sprt.Y  &&
        hit_rect.left   < work->Item.Sprt.X  &&
        hit_rect.right  > work->Item.Sprt.X  )
    {//アイテム取得時の処理
        switch(work->Item.ID)
        {//アイテムのIDに応じて処理を変える(ここではメッセージのみ)
            case ITEM_00:
                sprintf( work->DispMess,"スコア加算");
                break;
            case ITEM_01:
```

```
                        sprintf( work->DispMess,"パワーアップ");
                        break;
                    case ITEM_02:
                        sprintf( work->DispMess,"スピードアップ");
                        break;
                    case ITEM_03:
                        sprintf( work->DispMess,"1UP");
                        break;
                }
                //取得後アイテム処理は終了
                work->ItemFlag = false;
            }
            //アイテムの移動処理と表示
            work->Item.Sprt.X += work->Item.AddX;
            work->Item.Sprt.Y += work->Item.AddY;
            //画面外への移動でアイテムを消去
            if( work->Item.Sprt.Y > SCREEN_HEIGHT ) work->ItemFlag = false;
            //指定のアイテム色で描画
            EX09_01_CustomDraw( &work->Item.Sprt,1,work->Item.Color);
        }

        //自機の表示
        SpriteDraw( &work->MyShip,0);

        //取得したアイテム効果の表示
        g_pFont->DrawText( NULL, work->DispMess, -1, &font_pos, DT_LEFT,
0xffffffff);
}
```

9-2 キャラからアイテムを出す

アイテムを運んでくる

アイテムが出現する時、いきなり画面に出現するのではなく、特定のキャラクターが運んで来る事があります。

こういった処理は、通常のアイテム出現処理より手間はかかりますが、単純に出現させるより、見た目にもゲーム的にもより面白くなります。

ここでは、こういった特定のキャラがアイテムを落とす処理を作成してみましょう。

基本的な処理概念としては、[9－1]と同様、当たり判定とIDによるアイテム管理を行ないます。

ただ、[9－1]では単体のアイテム判定でしたが、今回は複数のキャラクターが出てくるため、タスクを使用してアイテムの作成を行なっています。

アイテムを運ぶキャラの作成

それでは実際のプログラムを解説していきます。サンプルは画面上部を移動するキャラが、一定時間ごとにボール（アイテム）を落とすものです。

キャラ作成

まず初期化でアイテムを落とすキャラの作成を行ないます。

これはタスクで作成され、処理関数EX09_02_enemy_moveで処理を行ないます。

処理内容は単純で、画面左へ移動しながら時間を計測し、一定時間が過ぎたらアイテムの処理タスクを生成します。

落とすアイテムの作成

この際に生成するアイテムはランダムで作成し、キャラから落下してる様に見せるため、座標をコピーしています。

あと、マクロで処理優先を定義し、アイテム処理タスクである事を示す様にしています。

なお、アイテムの処理そのものは落下するだけです。画面外へ移動する事で消え、特別な事はしていません。

アイテムの当たり判定処理

次にメイン処理を見ていきます。自機の移動処理後、当たり判定を行ないます。

当たり判定を行なうアイテムはタスクで管理されているため、チェックのために専用の関数EX09_02_get_itemを作成しています。

処理内容は、タスクを辿ってアイテム処理タスクを探し出し、個々のタスクと判定を行なうものです。

アイテム処理タスクには専用の処理優先番号が割り振られているため、容易に判断が出来ます。

判定後、アイテムと接触している場合は取得したものと判断し、アイテム処理タスクを消去後、アイテムの種別IDを返します。

なお、この関数の処理は第8章の当たり判定の処理を応用したもので、そちらでも解説しています。より詳細に知りたい方はそちらも参照してください。

そしてチェック関数の終了後、返ってきた値に応じて、アイテム処理を行ないます。

ここは、関数の返り値をそのまま、switch文に渡して処理しているので、アイテムIDを保持しておく場合は注意してください。

以上で処理は終了です。

LIST 9−2−1 キャラからアイテムを出す

```
#define NO_ITEM     -1
#define ITEM_00      0
#define ITEM_01      1
#define ITEM_02      2
#define ITEM_03      3
#define ITEM_MAX     4

#define ITEM_SPEED  5.0
#define MOVE_SPEED  4.0
#define ENEMY_SPEED 8.0
#define DROP_TIME   50
#define ENEMY_PRIORITY  0x2000
#define ITEM_PRIORITY   0x4000

typedef struct{
    SPRITE          MyShip;
    char            DispMess[64];
```

```
} EX09_02_STRUCT;

typedef struct{
    SPRITE          Sprt;
    int             Timer;
} EX09_02_ENEMY;

void EX09_02_item_move(TCB* thisTCB)
{
    EX09_01_ITEM* work = (EX09_01_ITEM*)thisTCB->Work;
    DWORD item_color[] =
{//アイテムの色
    0xff0000ff,
    0xff00ff00,
    0xffff0000,
    0xffffffff,
};

    //画面外に出たら消去
    if( work->Sprt.Y > SCREEN_HEIGHT )
    {//終了処理
        TaskKill(thisTCB);
        return;
    }
    //アイテムの色を決定
    work->Color  = item_color[ work->ID ];
    //アイテムの移動処理と表示
    work->Sprt.X += work->AddX;
    work->Sprt.Y += work->AddY;
    EX09_01_CustomDraw( &work->Sprt,1,work->Color);
}

void EX09_02_enemy_move( TCB* thisTCB )
{
    EX09_02_ENEMY* work = (EX09_02_ENEMY*)thisTCB->Work;
    TCB*    item_tcb;
    EX09_01_ITEM* item_work;

    if(work->Timer++ >= DROP_TIME)
    {//一定時間毎に、アイテムを落とす
```

```
            //落とすアイテムの作成
            item_tcb = TaskMake( EX09_02_item_move , ITEM_PRIORITY );
            item_work = (EX09_01_ITEM*)item_tcb->Work;
            //出現位置は、現在の座標をコピーする
            item_work->Sprt.X = work->Sprt.X;
            item_work->Sprt.Y = work->Sprt.Y;;
            item_work->AddY    = ITEM_SPEED;
            //アイテムの種類をランダムに決定する
            item_work->ID      = rand() % ITEM_MAX;

            //アイテムを落とす時間を初期化
            work->Timer = 0;
        }

         //画面外に出たら座標を戻す
        if( work->Sprt.X > SCREEN_WIDTH ) work->Sprt.X = 0;

        //移動処理と表示
        work->Sprt.X += ENEMY_SPEED;
        SpriteDraw( &work->Sprt,2);
    }

    int EX09_02_get_item( RECT* HitRect )
    {
        TCB*    check_tcb;
        EX09_01_ITEM* item_work;

        //タスクを全部チェックする
        //タスクの先頭アドレスを取得
        check_tcb = GetTaskTop();
        //タスクヘッドは処理せず飛ばす
        check_tcb = check_tcb->Next;
        //タスクヘッドにたどり着いたら１順
        while( check_tcb->Prio != 0x0000)
        {
            if( check_tcb->Prio == ITEM_PRIORITY )
            {//タスクがアイテムであれば、当たり判定をチェックする
                item_work = (EX09_01_ITEM*)check_tcb->Work;
                //対象と当たり判定の矩形を比較
                if( HitRect->top    < item_work->Sprt.Y &&
```

```
                HitRect->bottom > item_work->Sprt.Y &&
                HitRect->left   < item_work->Sprt.X &&
                HitRect->right  > item_work->Sprt.X )
            {//アイテムに接触した時の処理
                //アイテムを消去し、アイテムIDを返す
                TaskKill( check_tcb );
                return item_work->ID;
            }
        }
        //次にチェックするタスク
        check_tcb = check_tcb->Next;
    }
    return NO_ITEM;
}

void init09_02(TCB* thisTCB)
{
    EX09_02_STRUCT* work = (EX09_02_STRUCT*)thisTCB->Work;
    //使用するテクスチャの読み込み
    D3DXCreateTextureFromFile( g_pD3Ddevice, "..¥¥..¥¥data¥¥0044.png",&g_pTex[0] );
    D3DXCreateTextureFromFile( g_pD3Ddevice, "..¥¥..¥¥data¥¥0055.png",&g_pTex[1] );
    D3DXCreateTextureFromFile( g_pD3Ddevice, "..¥¥..¥¥data¥¥block1.png",&g_pTex[2] );

    //自機座標の初期化
    work->MyShip.X = SCREEN_WIDTH  / 2;
    work->MyShip.Y = SCREEN_HEIGHT / 2;
    //アイテムを落とすキャラの作成
    TaskMake( EX09_02_enemy_move , ENEMY_PRIORITY );
}

void exec09_02(TCB* thisTCB)
{
    EX09_02_STRUCT* work = (EX09_02_STRUCT*)thisTCB->Work;
    RECT hit_size_rect = { 0, 0, 64, 64, };
    RECT hit_rect;
    RECT font_pos = {  0,  0,640,480,};


    if( g_InputBuff & KEY_RIGHT ) work->MyShip.X += MOVE_SPEED;
    if( g_InputBuff & KEY_LEFT  ) work->MyShip.X -= MOVE_SPEED;
```

```
	//アイテム関連の処理

	//アイテム取得時の当たり判定矩形を計算
	hit_rect.top    =   work->MyShip.Y + hit_size_rect.top;
	hit_rect.bottom =   work->MyShip.Y + hit_size_rect.bottom;
	hit_rect.left   =   work->MyShip.X + hit_size_rect.left;
	hit_rect.right  =   work->MyShip.X + hit_size_rect.right;

	//アイテムの取得チェック
	switch( EX09_02_get_item(&hit_rect) )
	{//取得アイテムのIDに応じて処理を変える
		case ITEM_00: sprintf( work->DispMess,"スコア加算"); break;
		case ITEM_01: sprintf( work->DispMess,"パワーアップ"); break;
		case ITEM_02: sprintf( work->DispMess,"スピードアップ"); break;
		case ITEM_03: sprintf( work->DispMess,"１ＵＰ"); break;
		//アイテムの取得は無し
		default: break;
	}

	//自機の表示
	SpriteDraw( &work->MyShip,0);

	//取得したアイテム効果の表示
	g_pFont->DrawText( NULL, work->DispMess, -1, &font_pos, DT_LEFT,
0xffffffff);
}
```

9-3 撃つと出てくる隠しアイテム

隠しアイテム

何も見えない所にショットを当てて、アイテムを出現させる、いわゆる「隠しアイテム」ですが、実はこの処理は非常に簡単です。

通常の移動や敵の処理と同じように処理を行ない、ショットとの接触判定を行なって、アイテムを出現するようにするだけです。

もちろんこの際には表示は行なわず、当たり判定とアイテム出現の処理のみを行ないます。

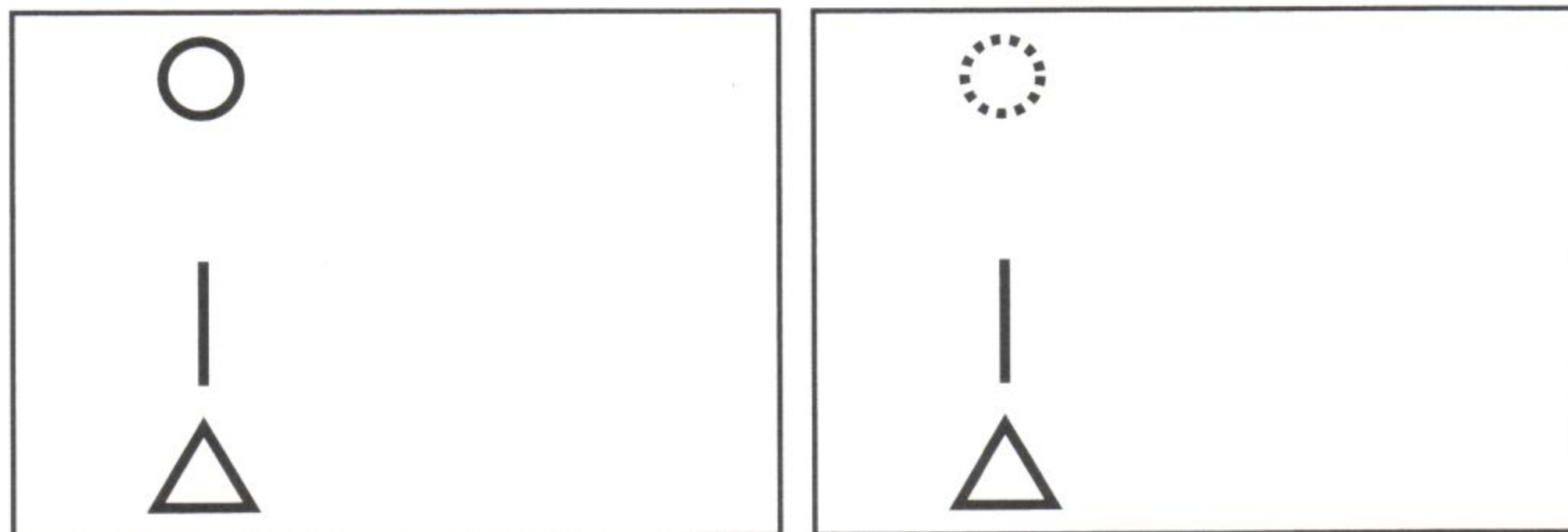
アイテム出現処理の応用

アイテム出現の際の処理は、現在までに解説したアイテム処理とほぼ同じなため割愛しますが、この処理概念自体は様々に利用できます。

例えば、背景と一緒に動く様にすれば※、背景の一部からアイテムが出現する様に見えますし、耐久力を保持するようにすれば、何発もショットを打ち込まないと出現しないようにうにする事が出来るでしょう。

ただし、隠しとはいえアイテムですので、あまり特殊な事をするとプレイヤーが出現をさせる事が難しくなります。

出現しない（出来ない）アイテムはバランスからいっても、あまり良い事とはいえませんので作成の際は気をつけるべきでしょう。

図9−3−1 隠しアイテムの処理は、通常の移動や処理として扱うイメージ図

隠しアイテムは表示の処理を行わないだけで、処理自体は、通常処理として扱う

※［6−4］スクロールに合わせてキャラを動かす参照。

9-4 アイテムの種類について

アイテムの種類とゲームバランス

アイテムの種類が及ぼす影響

ここでは、少しプログラムから離れてアイテムの種類について考えてみます。

アイテムというものは大きく分けて２種類に分けられます。

１つはプレイヤーにとって有利な効果のあるアイテムです。スコアアップやパワーアップがこれに該当します。

そしてもう１つは、プレイヤーにとって不利になるアイテム、いわゆるマイナスアイテムです。実はこちらの方がある意味でとても重要です。

なぜなら、プレイヤーにとっては、こちらの方が「ストレス」が高く、ゲームバランス上、とても危険な要素をはらんでいるからです。

マイナス要素は目立つ

もう少し平たく言うと、プレイヤーは「ゲームの進行を有利にする展開は当然」と無意識に考えており、マイナスの要素はそれだけで、プラスの要素より目立ってしまうのです。

例えば、フルパワーアップを行なうアイテムがあり、それに対になるアイテムとして、一切のパワーアップを無くすアイテムを作成したとします。

もちろん理屈の上では双方が同じ数だけ出てくれば、バランスは取れている事になります。

ですが、実際にこういった要素を組み込むと、マイナス要素だけが目立ってしまい、「とんでもないアイテムが出現する厳しいゲームだ」と思われてしまうでしょう。

結局最後はバランスが大事

ただし、これはあくまで極端な例です。ある程度やりこんだプレイヤーにとっては、こういったマイナスアイテムは、逆にゲーム上とてもよい刺激となり、一定の緊張感を与えてくれます。

上手くバランスを取って入れる事が肝要です。

9-5 アイテムの出現について

アイテムの出現率とバランス

ゲームバランスの話のついでという訳ではありませんが、アイテムの出現についても触れておきます。

アイテムの出現は、ゲームバランスにも強い影響を与えます。
例えば体力制のアクションゲームで、敵を倒すたびに、体力全回復のアイテムが出てきたらどうでしょうか？
多分、プレイヤーにとってはとても易しすぎてすぐに飽きてしますでしょう。
ここまで極端でなくても、アイテムの数がとても少なくて、いわゆる「難所」を越えられなかったり、ボスを倒した直後に無敵になるアイテムが出る等、バランス的に考えさせられるアイテム出現の例は枚挙に暇がありません。

アイテム出現位置の固定

そこで、こういった難易度のバランスを取るため、アクションゲーム等では出現するアイテムの数や位置は、ある程度固定される事が多いようです。
ただし、単純にスクロール位置や、時間などで出現してしまうと、展開の予想が出来やすいため、ゲーム性が下がる事があります。

これは考え方によっては、攻略を立てやすくしているともいえるので一概に下がってるとは言えないのですが、プレイヤーに取ってどの状態が一番面白いか？　を常に考えて配置を行なう必要があります。

9-6 一定時間効果を発揮するアイテム

時間で効果を発揮するアイテム

一定時間効果を発揮するアイテムについて考えてみます。

一定時間の強力なパワーアップや一定時間の無敵等、ゲームでは時間限定で効果を発揮するアイテムが多々あります。

また、時間以外でも、例えば耐久力が低い時だけ効果を発揮するアイテム等、特殊な条件や期間のみ効果を発揮するアイテムもあります。

こういった時間や特定条件でのみ持続するアイテムは、1つや2つ程度でしたら良いのですが、数が増えると管理が大変になります。

そこで通常このようなアイテム処理はタスクなどの並列動作を用いて管理します。

並列動作を用いて管理を行なえば、時間だけではなく様々な条件でのアイテム処理が可能になります。

即時処理のアイテムと並列処理のアイテム

それでは実際のプログラムを解説します。サンプルは、アイテムを取得した瞬間一定時間メッセージを表示する物です。

なお、このプログラムはアイテムの作成部分を含めた処理の大部分に[9－2]の処理プログラムを用いて再利用しています。

ここでは、メインの解説のみを行ないますので、作成部分等の詳細はそちらを参照してください。

処理の解説

ではメイン処理部分の解説です。自機の移動後、当たり判定を行なう部分までは、[9－2]と同じです。

その後判定をチェックし、取得したアイテムに合わせて並列処理を作成します。

もちろんこの時、特殊な条件を要しないアイテムであれば即時アイテム処理を行なって問題ありません。

ここでは、アイテムのID 1と2のみ並列処理(タスク)の作成を行なっています。

タスクはIDにあわせて、処理関数EX09_06_item_effect01とEX09_06_item_effect02を割り当てられます。

プログラムの内容

実際の処理内容ですが、両方のタスク処理関数とも、一定時間、処理メッセージを表示するだけのものです。

もちろん実際は表示だけでなく、処理を行なう必要がありますので、内容に合わせて適宜処理関数の中身を変更すると良いでしょう。

LIST 9－6－1 アイテム処理:一定時間効果を発揮

```
#define NO_ITEM     -1
#define ITEM_00     0
#define ITEM_01     1
#define ITEM_02     2
#define ITEM_03     3
#define ITEM_MAX    4

#define MOVE_SPEED  4.0
#define ENEMY_PRIORITY  0x2000
#define EFFECT_PRIORITY 0x3000
#define EFFECT_TIME 30

typedef struct{
    SPRITE          MyShip;
    char            DispMess[64];
} EX09_06_STRUCT;


void EX09_06_item_effect01( TCB* thisTCB )
{
    RECT font_pos = {  0, 32,640,480,};

    //一定時間効果を発揮した後消去
    if( thisTCB->Work[0]++ >= EFFECT_TIME ) TaskKill(thisTCB);

    g_pFont->DrawText( NULL, "パワーアップ中!!", -1, &font_pos, DT_LEFT,
0xffffffff);
}

void EX09_06_item_effect02( TCB* thisTCB )
{
    RECT font_pos = {  0, 32,640,480,};
```

```
    //一定時間効果を発揮した後消去
    if( thisTCB->Work[0]++ >= EFFECT_TIME ) TaskKill(thisTCB);

    g_pFont->DrawText( NULL, "スピードアップ中!!", -1, &font_pos, DT_LEFT,
0xffffffff);
}

void init09_06( TCB* thisTCB )
{
    EX09_06_STRUCT* work = (EX09_06_STRUCT*)thisTCB->Work;
    //使用するテクスチャの読み込み
    D3DXCreateTextureFromFile( g_pD3Ddevice, "..¥¥..¥¥data¥¥0044.png",&g_pTex[0] );
    D3DXCreateTextureFromFile( g_pD3Ddevice, "..¥¥..¥¥data¥¥0055.png",&g_pTex[1] );
    D3DXCreateTextureFromFile( g_pD3Ddevice, "..¥¥..¥¥data¥¥block1.png",&g_pTex[2] );

    //自機座標の初期化
    work->MyShip.X = SCREEN_WIDTH  / 2;
    work->MyShip.Y = SCREEN_HEIGHT / 2;
    //アイテムを落とすキャラの作成
    TaskMake( EX09_02_enemy_move , ENEMY_PRIORITY );
}

void exec09_06( TCB* thisTCB )
{
    EX09_06_STRUCT* work = (EX09_06_STRUCT*)thisTCB->Work;
    RECT hit_size_rect = { 0, 0, 64, 64, };
    RECT hit_rect;
    RECT font_pos = {  0, 32,640,480,};


    if( g_InputBuff & KEY_RIGHT ) work->MyShip.X += MOVE_SPEED;
    if( g_InputBuff & KEY_LEFT  ) work->MyShip.X -= MOVE_SPEED;

    //アイテム関連の処理

    //アイテム取得時の当たり判定矩形を計算
    hit_rect.top    =   work->MyShip.Y + hit_size_rect.top;
    hit_rect.bottom =   work->MyShip.Y + hit_size_rect.bottom;
    hit_rect.left   =   work->MyShip.X + hit_size_rect.left;
```

```
    hit_rect.right  =   work->MyShip.X + hit_size_rect.right;

    //アイテムの取得チェック
    switch( EX09_02_get_item(&hit_rect) )
    {//時間限定の処理はタスクを作成して対応
        case ITEM_00: sprintf( work->DispMess,"スコア加算"); break;
        case ITEM_01:
                //アイテム効果用のタスクを作成
                sprintf( work->DispMess,"");
                TaskMake( EX09_06_item_effect01, EFFECT_PRIORITY);
            break;
        case ITEM_02:
                //アイテム効果用のタスクを作成
                sprintf( work->DispMess,"");
                TaskMake( EX09_06_item_effect02, EFFECT_PRIORITY);
            break;
        case ITEM_03: sprintf( work->DispMess,"１ＵＰ"); break;

        //アイテムの取得は無し
        default: break;
    }

    //自機の表示
    SpriteDraw( &work->MyShip,0);

    //取得したアイテム効果の表示
    g_pFont->DrawText( NULL, work->DispMess, -1, &font_pos, DT_LEFT, 0xffffffff);
}
```

9-7 取得後に使用して効果を発揮するアイテム

所持アイテム

ゲームによっては、アイテムの取得直後に効果を発揮するのではなく、一時的にアイテムを保持しておきゲームの状況に応じて使用するいわゆる「所持アイテム」があります。

こういったアイテムは、操作性の向上だけでなく、ゲーム性を高める上でも非常に有効な手法です。

ここでは、この所持アイテムの処理を作成してみましょう。

処理概念ですが、実は通常のアクションにおけるアイテム処理と殆ど変わりはありません。

ただ、アイテムの保持を行なうためにバッファを設け、所持しているアイテムを管理する処理が必要となります。

アイテムの取得、保持

それではプログラムを見ていきましょう。サンプルは、取得したアイテムを保持しておき、Z キーで使用するものです。

なおこのプログラムは前項と同様に、処理の大半を[9－2]の処理プログラムを用いて再利用しています。

メインの解説のみを行ないますので、その他の部分はそちらを参照してください。

ではメインの解説です。はじめに自機を移動し、当たり判定のチェックを行ないます。

チェック後、アイテムを取得していたら、所持しているアイテムの ID を変数 HaveItem に格納し、所持しているアイテムを変更します。

ここでは強制的に所持するようにしていますが、もし条件によって、アイテムの所持を行なわない場合は、この部分を変更してください。

その後、所持アイテムの ID を使用し、所持しているアイテムの種類を表示します。

以上で保持の処理は終了です。

アイテムの使用

次に実際にアイテムの使用を行なう処理です。

この部分は、単純で HaveItem に格納されている、所持アイテムに応じて処理を行なうだけです。

ただし、アイテムの使用後に、所持アイテムを破棄する事を忘れないで下さい。

また、もし特殊な効果を付加するのであれば、[9－6]の様に並列処理を作成しても良いでしょう。

LIST 9－7－1 アイテムを取得後に使用する

```
#define NO_ITEM     -1
#define ITEM_00     0
#define ITEM_01     1
#define ITEM_02     2
#define ITEM_03     3
#define ITEM_MAX    4

#define MOVE_SPEED  4.0
#define ENEMY_PRIORITY  0x2000
#define EFFECT_PRIORITY 0x3000
#define EFFECT_TIME 30

typedef struct{
    SPRITE          MyShip;
    int             HaveItem;
    char            DispMess[64];
} EX09_07_STRUCT;


void init09_07( TCB* thisTCB )
{
    EX09_07_STRUCT* work = (EX09_07_STRUCT*)thisTCB->Work;
    //使用するテクスチャの読み込み
    D3DXCreateTextureFromFile( g_pD3Ddevice, "..¥¥..¥¥data¥¥0044.png",&g_pTex[0] );
    D3DXCreateTextureFromFile( g_pD3Ddevice, "..¥¥..¥¥data¥¥0055.png",&g_pTex[1] );
    D3DXCreateTextureFromFile( g_pD3Ddevice, "..¥¥..¥¥data¥¥block1.png",&g_pTex[2] );

    //自機座標の初期化
    work->MyShip.X = SCREEN_WIDTH  / 2;
    work->MyShip.Y = SCREEN_HEIGHT / 2;
    //アイテムを落とすキャラの作成
    TaskMake( EX09_02_enemy_move , ENEMY_PRIORITY );
}

void exec09_07( TCB* thisTCB )
```

```
{
    EX09_07_STRUCT* work = (EX09_07_STRUCT*)thisTCB->Work;
    int  check_item;
    RECT hit_size_rect = { 0, 0, 64, 64, };
    RECT hit_rect;
    RECT font_pos = {  0, 32,640,480,};


    if( g_InputBuff & KEY_RIGHT )
        work->MyShip.X += MOVE_SPEED;
    if( g_InputBuff & KEY_LEFT  )
        work->MyShip.X -= MOVE_SPEED;

    //アイテム関連の処理

    //アイテム取得時の当たり判定矩形を計算
    hit_rect.top    =   work->MyShip.Y + hit_size_rect.top;
    hit_rect.bottom =   work->MyShip.Y + hit_size_rect.bottom;
    hit_rect.left   =   work->MyShip.X + hit_size_rect.left;
    hit_rect.right  =   work->MyShip.X + hit_size_rect.right;

    //アイテムの取得チェック
    check_item = EX09_02_get_item(&hit_rect);
    if( check_item != NO_ITEM )
    {//アイテムを取得していたら、所持アイテムを切り替える
        work->HaveItem = check_item;
        switch( check_item )
        {//取得したアイテムに応じて、所持アイテムの表示をする
            case ITEM_00:
                sprintf(work->DispMess,"所持ITEM:スコア加算");
                break;
            case ITEM_01:
                sprintf(work->DispMess,"所持ITEM:パワーアップ");
                break;
            case ITEM_02:
                sprintf(work->DispMess,"所持ITEM:スピードアップ");
                break;
            case ITEM_03:
                sprintf(work->DispMess,"所持ITEM:1UP");
                break;
```

```
        }
    }

    if( g_DownInputBuff & KEY_Z )
    {//Zキーで所持アイテムの使用
        switch( work->HaveItem )
        {//アイテムのIDに応じて処理を変える(ここではメッセージのみ)
            case ITEM_00:
                sprintf( work->DispMess,"スコア加算ITEM使用");
                break;
            case ITEM_01:
                sprintf( work->DispMess,"パワーアップITEM使用");
                break;
            case ITEM_02:
                sprintf( work->DispMess,"スピードアップITEM使用");
                break;
            case ITEM_03:
                sprintf( work->DispMess,"1UP ITEM使用");
                break;
            //アイテムを所持していない
            default:
                sprintf(work->DispMess,"ITEMを所持していません");
                break;
        }
        //所持アイテムの破棄
        work->HaveItem = NO_ITEM;
    }

    //自機の表示
    SpriteDraw( &work->MyShip,0);

    //取得したアイテム効果の表示
    g_pFont->DrawText( NULL, work->DispMess, -1, &font_pos, DT_LEFT,
0xffffffff);
}
```

9-8 多数のアイテムの管理

多数のアイテムの管理

ここでは、多数のアイテムを管理する手法について考えてみます。

RPG等におけるアイテムは、シューティングのアイテムなどとは違って、扱う数が非常に多くなります。

前項までの手法は処理を中心とした物であり、少数であれば問題はないのですが、多数のアイテムを管理するには向いていません。

こういった多数のアイテムを管理するには、データによる管理が必要になってきます。

アイテム管理の構造体

具体的にはIDだけではなく、アイテム専用の構造体を用意して処理の共通化を図ったり、特別な状態をデータとして持つようにします。

以降の項目では、サンプルとして以下の構造体を元に解説を行ないますが、その前にデータの内容について簡単に触れておきます。


```
typedef struct _EX09_RPG_ITEM_DATA {
    //処理関数
    int (*ItemFunc)( _EX09_RPG_ITEM_DATA*, char* );
    //アイテム名
    char*           ItemName;
    //処理タイプ
    int             ItemType;
    //アイテム特有のデータ
    int             Param;
} EX09_RPG_ITEM_DATA;
```

まずこのアイテムの処理関数ItemFuncを定義します。この関数は引数として、このデータへのポインタと、外部への出力を行なうポインタを受け取ります。

アイテム名ItemNameは、そのまま、表示されるアイテム名となります。

処理タイプItemTypeは、定義されるアイテムの基本的なグループ分けをするもので、基本的な処理を区別するために用いられます。

最後はアイテム特有のデータを示す物で、具体的には同じ処理関数を指定した時に、パラメータ等として使用されます。

データ内容については以上です。次項からはより詳細な解説を行なっていきますが、データ定義のサンプルとして簡単なメニューの表示例を示しておきます。

LIST 9－8－1 アイテム処理の管理

```
#define ITEM_NORMAL     0
#define ITEM_EVENT      1

//メッセージの高さ
#define FONT_SIZE 16
#define NO_ITEM    -1
#define HAVE_MAX  4
#define ITEM_X    192
#define ITEM_Y    64

typedef struct{
    int            HaveItem[HAVE_MAX];
} EX09_08_STRUCT;

static EX09_RPG_ITEM_DATA g_EX09_08_ItemData[] =
{
  {NULL,"薬草"      , ITEM_NORMAL, 5,},  //ID00
  {NULL,"治療薬"    , ITEM_NORMAL,50,},  //ID01
  {NULL,"毒消し草"  , ITEM_NORMAL, 0,},  //ID02
  {NULL,"復活の薬"  , ITEM_NORMAL, 0,},  //ID03
};

void init09_08(TCB* thisTCB)
{
    EX09_08_STRUCT* work = (EX09_08_STRUCT*)thisTCB->Work;

    //所持用の配列にアイテムを格納
    work->HaveItem[0] = 0;
    work->HaveItem[1] = 1;
    work->HaveItem[2] = 2;
    work->HaveItem[3] = 3;
}
```

```
void exec09_08(TCB* thisTCB)
{
    EX09_08_STRUCT* work = (EX09_08_STRUCT*)thisTCB->Work;
    int loop;
    RECT font_pos = { 0, 0, 640, 480, };


    //アイテム内容の表示
    font_pos.left = ITEM_X;
    font_pos.top  = ITEM_Y;
    for(loop = 0; loop < HAVE_MAX; loop++)
    {
        g_pFont->DrawText( NULL,
            g_EX09_08_ItemData[ work->HaveItem[ loop ] ].ItemName,
            -1,
            &font_pos,
            DT_LEFT,
            0xffffffff);
        font_pos.top += FONT_SIZE;
    }
}
```

プログラムはマクロで指定した座標に定義したアイテム名で、一覧を表示するものです。

アイテム処理等も行なっていませんが、アイテムの所持をタスク上の配列変数 HaveItem で行なっています。

9-9 アイテム処理の効果反映

アイテム処理の流れ

では実際のアイテム処理を解説していきます。プログラムは、方向キーでメニューのアイテムを選択し、Z キーで使用するものです。

まず初期化として、アイテム所持を管理する配列HaveItem にアイテムのID を代入します。

次にメインの処理として、アイテムの選択と決定処理を行ないます。

選択処理は方向キーを判断して、選択中の配列のID（変数SelectMenu）を管理しています。

決定処理は、選択された配列のID を元にアイテムのID を取得し、そのID のデータを処理する流れで行なっています。

実際のアイテム処理

実際のアイテム処理は、アイテムのデータ項目ItemFunc で定義された処理関数で処理されます。

この関数は、入力としてアイテムが定義されたデータポインタと、結果を反映させる出力へのポインタを受け取ります。

今回のサンプルでは、効果の反映はメッセージで行なっているため、文字列バッファへのポインタを受け取るようにしています。

実際に使用する際は、通信用の構造体を用意してそのポインタを受け渡すようにすると良いでしょう。

処理後、使用されたアイテムは消去処理を行ないます。

ただし、ここで消去されるアイテムは一般的なアイテムだけで、特別なアイテム（イベントアイテム等）は行なわないようにしています。

特殊なアイテムかどうかは、データ項目のItemType で定義、判断します。

最後に、アイテムの一覧とメッセージの表示処理を行なっています。

LIST 9 －9－1

```
#define ITEM_NORMAL     0
#define ITEM_EVENT      1

typedef struct{
```

```
    int             HaveItem[HAVE_MAX];
    SPRITE          SelectMark;
    int             SelectMenu;
    char            DispMess[64];
} EX09_09_STRUCT;


int EX09_09_use_item00( EX09_RPG_ITEM_DATA* ItemData ,char* ResMess )
{//薬草・治療薬等、回復全般

   //回復処理を行なう(処理はサンプルでは例示しません)

    //メッセージを返す
    sprintf( ResMess,"%sを使用しました。HP %d回復！！",ItemData->ItemName,
ItemData->Param );
}

int EX09_09_use_item02( EX09_RPG_ITEM_DATA* ItemData ,char* ResMess )
{//毒消し草
    //メッセージを返す
    sprintf( ResMess,"毒消し草を使用しました。毒が体から消えた！！");
}

int EX09_09_use_item03( EX09_RPG_ITEM_DATA* ItemData ,char* ResMess )
{//黄金の鍵　イベントアイテム
    //メッセージを返す
    sprintf( ResMess,"このアイテムを使用する事は出来ません！");
}

static EX09_RPG_ITEM_DATA g_EX09_09_ItemData[] =
{
  {EX09_09_use_item00,"薬草"      , ITEM_NORMAL, 5,},  //ID00
  {EX09_09_use_item00,"治療薬"    , ITEM_NORMAL,50,},  //ID01
  {EX09_09_use_item02,"毒消し草"  , ITEM_NORMAL, 0,},  //ID02
  {EX09_09_use_item03,"黄金の鍵"  , ITEM_EVENT , 0,},  //ID03
};

void init09_09(TCB* thisTCB)
{
```

```
    EX09_09_STRUCT* work = (EX09_09_STRUCT*)thisTCB->Work;

    //使用するテクスチャの読み込み
    D3DXCreateTextureFromFile( g_pD3Ddevice, "..¥¥..¥¥data¥¥select_mark.
png",&g_pTex[0] );

    //所持用の配列にアイテムIDを格納(アイテムの取得)
    work->HaveItem[0] = 0;
    work->HaveItem[1] = 1;
    work->HaveItem[2] = 2;
    work->HaveItem[3] = 3;
}

void exec09_09(TCB* thisTCB)
{
    EX09_09_STRUCT* work = (EX09_09_STRUCT*)thisTCB->Work;
    int select_item;
    int loop;
    RECT font_pos = { 0, 0, 640, 480, };

    //アイテム選択
    if( g_DownInputBuff & KEY_DOWN )
    {
        work->SelectMenu++;
        if( work->SelectMenu >= HAVE_MAX )work->SelectMenu = 0;
    }
    if( g_DownInputBuff & KEY_UP  )
    {
        work->SelectMenu--;
        if( work->SelectMenu < 0 ) work->SelectMenu = HAVE_MAX-1;
    }

    if( g_DownInputBuff & KEY_Z  )
    {//Zキーで選択中のアイテムの使用
        //選択中の位置にアイテムがあるかを確認
        select_item = work->HaveItem[ work->SelectMenu ];

        if( select_item != NO_ITEM )
        {//アイテムがあるときのみ実行
```

```
                //アイテム処理関数を実行
                //データとメッセージのポインタを渡す
                g_EX09_09_ItemData[ select_item ].ItemFunc(
                    &g_EX09_09_ItemData[ select_item ],
                    work->DispMess);
            }

            //通常のアイテムは使用後に消去する
            if( g_EX09_09_ItemData[ select_item ].ItemType == ITEM_NORMAL)
                work->HaveItem[ work->SelectMenu ] = NO_ITEM;
        }

        //選択マークの位置を決定
        //アイテム内容の表示

        work->SelectMark.X = ITEM_X - FONT_SIZE;
        work->SelectMark.Y = ITEM_Y + work->SelectMenu * FONT_SIZE;
        //選択マークの表示
        SpriteDraw(&work->SelectMark,0);

        //メニューの表示
        font_pos.left = ITEM_X;
        font_pos.top  = ITEM_Y;
        for(loop = 0; loop < HAVE_MAX; loop++)
        {
            if( work->HaveItem[ loop ] != NO_ITEM )
            {//所持アイテムがある時のみ表示
                g_pFont->DrawText(
                    NULL,
                    g_EX09_09_ItemData[ work->HaveItem[ loop ] ].ItemName,
                    -1,
                    &font_pos,
                    DT_LEFT,
                    0xffffffff);
            }
            font_pos.top += FONT_SIZE;
        }

        //アイテム使用時のメッセージの表示
        font_pos.top  = 448;
```

```
    font_pos.left = 32;
    g_pFont->DrawText( NULL, work->DispMess, -1, &font_pos, DT_LEFT, 0xffffffff);
}
```

9-10 所持アイテムの管理

所持アイテムの管理

実際にゲーム作成を進めていくと、アイテムの所持や破棄を頻繁に繰り返すようになります。

ここでは、それらの管理を行なうための関数を紹介します。

ここで紹介する関数は——

・アイテムの追加を行なう関数	**EX09_10_item_add**
・アイテムの削除を行なう関数	**EX09_10_item_del**
・特定アイテムの検索を行なう関数	**EX09_10_item_search**
・アイテムの並びの隙間を埋める	**EX09_10_item_correct**

——の４つです。

基本的にこれらの関数は、アイテム所持の管理として行なわれる為、所持の為の管理配列HaveItemに対して行なわれる事が前提となります。

各関数の解説

では、各関数を見ていきましょう。

アイテムの追加を行なう関数

まず、EX09_10_item_addです。

この関数は、引数に、管理配列のポインタと、追加するアイテムのIDを渡します。

プログラムは、まずアイテムをこれ以上持てるかどうか、配列の空きをチェックします。

もし配列に空きが無く、これ以上持てない様なら、エラーを返して処理は終了します。

もし持てる様なら、発見された空きの配列に、追加するIDを登録し、返り値として登録した配列のIndex値を返します。

アイテムの削除を行なう関数

次に、EX09_10_item_delです。

この関数はとてもシンプルで、指定された管理配列の中に、アイテムが無い状態である事を示すID NO_ITEMを代入するだけです。

特定アイテムの検索を行なう関数

次は、EX09_10_item_search です。この関数も引数に配列と検索を行なうIDを渡します。

処理はシンプルなループで、指定のIDが配列内にあるかどうかをチェックし、発見したらそのIDのINDEX番号を返します。

アイテムの並びの隙間を埋める

最後に、EX09_10_item_correct です。この関数はならんだアイテムの隙間を詰める関数です。

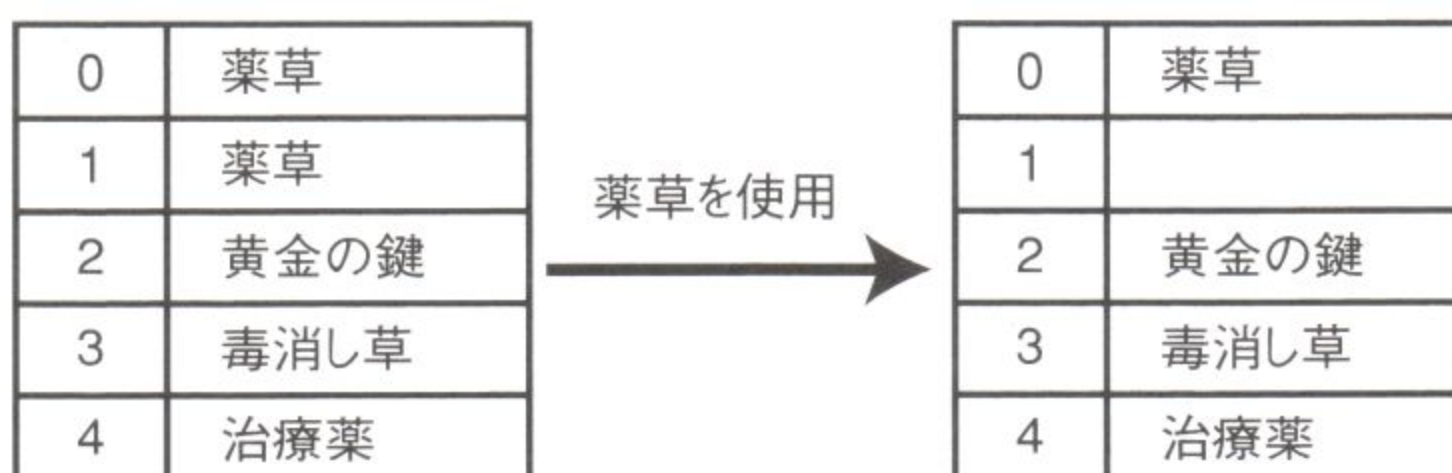
処理内容ですが、まず、アイテムの空きの欄を探し出します。そして、空いた欄以降の全ての欄を1つづつ詰めていきます。

この時ループだけでは、一番最後の欄は詰められないため、アイテム無しのID NO_ITEMを追加します。

図9－10－1 アイテム欄を1つづつ詰めるイメージ図

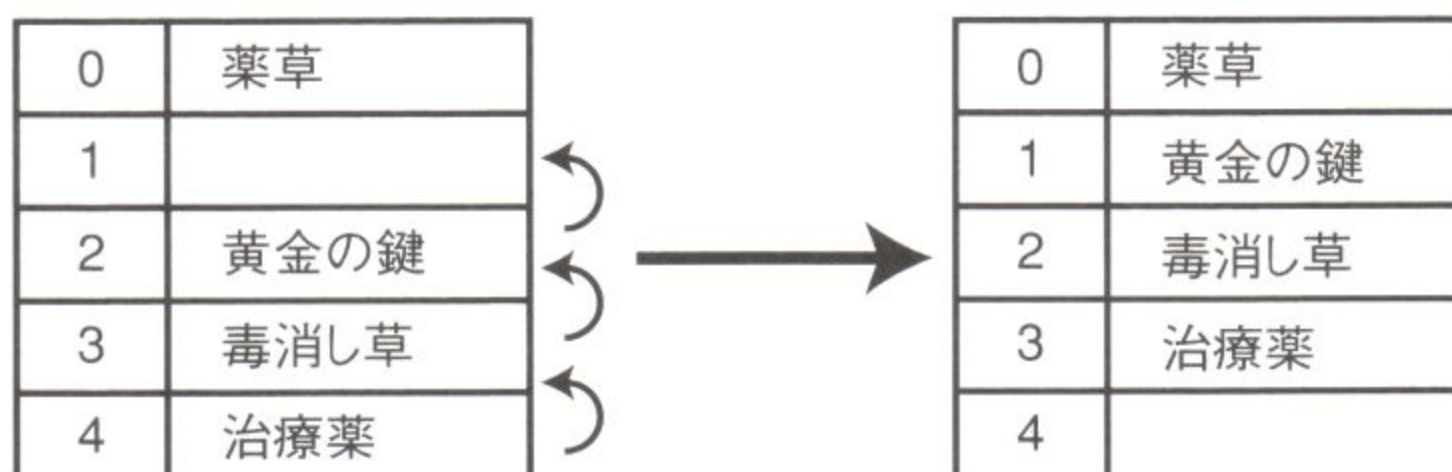
0	薬草
1	薬草
2	黄金の鍵
3	毒消し草
4	治療薬

薬草を使用 →

0	薬草
1	
2	黄金の鍵
3	毒消し草
4	治療薬

アイテムが使用されて消えるとアイテム欄に空きが出来る

0	薬草
1	
2	黄金の鍵
3	毒消し草
4	治療薬

→

0	薬草
1	黄金の鍵
2	毒消し草
3	治療薬
4	

その為、アイテム欄の空きをずらして埋める処理が必要

サンプルプログラム

最後にサンプルプログラムを解説します。処理概要はXキーでランダムにアイテムを取得し、Zキーで使用します。

なお、基本的には前項[9－9]の処理とほぼ同じなため、ここでは、違う部分のみを解説します。

アイテムの追加

まずアイテムの追加ですが、ランダムにアイテムを決めた後に、先ほど解説した関数EX09_10_item_addを用いて、アイテムを管理配列HaveItemに追加します。

この時、もし追加できないようなら、その旨をメッセージ用のバッファに出力します。

```
if( g_DownInputBuff & KEY_X  )
{//Xキーでランダムにアイテムを取得
    //アイテムをランダムに決める
    //(鍵は要素から外す)
    get_item = rand() % (HAVE_MAX - 1);

    //アイテムを追加
    err = EX09_10_item_add( work->HaveItem, get_item);
    if(err == -1)
    {//追加できなかったらメッセージを出す
        sprintf( work->DispMess,"アイテムを追加できませんでした。");
    }
}
```

アイテムの使用

後はアイテムの使用ですが、アイテムの使用処理自体は[9－9]とほぼ同じです。実際に関数も[9－9]の処理を再利用しています。

ただし、消去には、関数EX09_10_item_delを用い、消去後に空いたアイテムの並びの隙間を埋めるため、EX09_10_item_correctを呼び出していますので、そこだけ注意してください。

```
//アイテムの使用処理は[9-9]と同一

//アイテム使用後の処理
if( g_EX09_09_ItemData[ select_item ].ItemType == ITEM_NORMAL)
{//通常のアイテムは使用後に消去する
    EX09_10_item_del(work->HaveItem,work->SelectMenu );
   //消去後、開いた欄を詰める
    EX09_10_item_correct(work->HaveItem);
}
```

LIST 9 －10－1

```
#define ITEM_NORMAL     0
#define ITEM_EVENT      1

typedef struct{
    int          HaveItem[HAVE_MAX];
    SPRITE       SelectMark;
```

```
    int             SelectMenu;
    char            DispMess[64];
} EX09_10_STRUCT;


int EX09_10_item_add( int* ItemList,int ID )
{
    int loop;

    for(loop = 0; loop < HAVE_MAX; loop++)
    {//アイテム欄の空きを探す
        if( ItemList[ loop ] == NO_ITEM )break;
    }
    //アイテムをこれ以上もてなかったらエラー
    if( loop == HAVE_MAX ) return -1;
    //アイテムを追加
    ItemList[ loop ] = ID;
    //成功したらアイテムを挿入した位置を返す
    return loop;
}

void EX09_10_item_del( int* ItemList,int ID )
{
    //アイテムを削除
    ItemList[ ID ] = NO_ITEM;
}

int EX09_10_item_search( int* ItemList,int ID )
{
    int loop;

    for(loop = 0; loop < HAVE_MAX; loop++)
    {//指定IDのアイテムを探す
        if( ItemList[ loop ] == ID )break;
    }

    //アイテムを発見できなかったら終了
    if( loop == HAVE_MAX ) return -1;

    //発見したら、アイテムのIDを返す
```

```
    return loop;
}

void EX09_10_item_correct( int* ItemList )
{
    int loop;
    int collect_flag = 0;

    //初めのアイテム欄が空きかチェック
    if( ItemList[0] == NO_ITEM )collect_flag = true;
    for(loop = 0; loop < HAVE_MAX-1; loop++)
    {
        if(collect_flag)
        {//補正フラグが立った後のアイテムは
         //全て1個詰める
            ItemList[ loop ] = ItemList[ loop+1];
        }
        //空のアイテム欄を探す
        if( ItemList[ loop+1] == NO_ITEM )collect_flag = true;
    }

    //補正フラグが立っていたら、最後の欄はアイテム無し
    if(collect_flag) ItemList[ loop ] = NO_ITEM;
}
```

9-11 複数所持可能なアイテム

アイテムを複数持つ

ここでは複数所持できるアイテムを考えてみます。

ゲームによっては体力回復等のアイテムは、消耗品として非常に多数使用する場合があります。

こうした場合、アイテム欄1つで1つのアイテムしか持てない場合、すぐに所持数が最大値に達してしまい、非常に不便です。

そこで通常は、複数所持可能なアイテムを設定し、アイテム欄を圧迫しないようにします。

処理概要ですが、新たにアイテムのタイプとして複数所持可能なタイプを設定し、カウント処理を行なうようにしてやります。

ただしこの際に、カウント用の配列をアイテム欄と同じ数だけ、新たに設けてやらなくてはなりません。

```
//アイテム欄と同じ数だけカウント用の配列を持つ
typedef struct{
    int             HaveItem[HAVE_MAX];
    int             ItemCount[HAVE_MAX];
    SPRITE          SelectMark;
    int             SelectMenu;
    char            DispMess[64];
} EX09_11_STRUCT;
```

複数所持アイテムのプログラム

では実際の処理を見ていきます。内容は、方向キーで選択したアイテムをZキーで使用するもので、[9－10]とほぼ同じになります。

その為ここでは、実際の処理となるアイテム使用の部分とデータ内容だけ解説します。

データ内容

まずデータですが、定義自体はほぼ同じで、処理関数自体も同じです。ただし、複数アイテムであることを示す為、アイテムタイプに識別用の定数 ITEM_COUNT を定義しています。

```
static EX09_RPG_ITEM_DATA g_EX09_11_ItemData[] =
{
  {EX09_09_use_item00,"薬草"       , ITEM_COUNT , 5,},  //ID00
  {EX09_09_use_item00,"治療薬"     , ITEM_NORMAL,50,},  //ID01
  {EX09_09_use_item02,"毒消し草"   , ITEM_NORMAL, 0,},  //ID02
  {EX09_09_use_item03,"黄金の鍵"   , ITEM_EVENT , 0,},  //ID03
};
```

処理内容

次に処理ですが、先に説明したように、定義したアイテムのタイプによって、処理を分けています。

通常のアイテムは、処理した瞬間に消去を行なっていますが、複数所持可能なアイテムの場合は、先にバッファにある所持数を減らし、その数が0以下になった場合のみ消去しています。

アイテムの取得時は、逆にバッファ内の所持数を増やしてやるようにすればOKです。

LIST 9－11－1

```
//アイテムの使用処理
#define ITEM_NORMAL     0
#define ITEM_EVENT      1
#define ITEM_COUNT      2

switch(g_EX09_11_ItemData[ select_item ].ItemType)
{//アイテムのタイプによって処理を分ける
    case ITEM_NORMAL:
            //通常のアイテムは使用後に消去する
            EX09_10_item_del(work->HaveItem,work->SelectMenu );
            //消去後、開いた欄を詰める
            EX09_10_item_correct(work->HaveItem);
            break;
    case ITEM_COUNT:
            //カウントアイテムは数を減らしてチェック後消去する
            if( --work->ItemCount[ work->SelectMenu ] <= 0)
            {
                EX09_10_item_del(work->HaveItem,work->SelectMenu );
                //消去後、開いた欄を詰める
                EX09_10_item_correct(work->HaveItem);
            }
```

```
            break;
    default: break;
}
```

使用回数制限のあるアイテム

最後に、ここでは複数持つというイメージで処理を解説しましたが、まったく同じ処理で、使用回数制限のあるアイテムにする事も出来ます。

ただし、両方のアイテムが混在する場合は、プレイヤーが分かりにくくなるので、表示方法に工夫が必要になります。サンプルファイルを参照してみてください。

9-12 使用キャラ別の所持アイテム処理の管理

アイテムを使うキャラを増やすときの注意点

ここまでの処理では、使用キャラは１人として扱ってきました。

しかし実際のゲームでは、パーティ等として複数のキャラを同時に使用し、それぞれのキャラ毎にアイテムを所持しています。

これらの管理自体は、管理配列をキャラクター人数分だけ用意してやれば済むので、さほど問題はありません。

実際には、それらを管理する関数の作成の方が面倒になります。

例えば、複数所持できないアイテムを持った場合、キャラクター全員のアイテムをチェックする必要があるため、全チェック用の関数を作る必要があります。

アイテム種類の仕様は最初に決めておこう

また、通常のアイテムとは別の枠として、イベントアイテムを持つ場合等、ゲームの仕様によっては管理の手法自体がガラリと変わってしまう場合もあります。

そのため、ゲーム作り後半になって、問題が起きないように、何処まで柔軟に対応できるか？等の仕様は最初のうちに十分検討する必要があります。

Chapter 10

ゲームバランス

10-1 ゲームバランスについて

ゲームにおいてバランスの調整は非常に重要な項目です。

もしゲームが、企画の意図する通りに作成されたとしたら、そのゲームが面白いかどうかを決定づけるのは、バランス調整によるといっても過言ではありません。

ところが、ゲームバランスの定義というものは、非常にあいまいで不確かなものです。

その上、ゲームシステムや、難易度の仕組みによっても変わり、場合によってはゲームバランスの概念が存在しないゲームもあります。

実際、ゲームバランスの調整方法はゲームの数だけあると言ってもよく、プログラムを組むにしても、中々パターン化しにくいのが実情です。

とはいえ、ある程度の定石というものはあります。

向き不向きや、どれだけゲームを作りこみたいかによっても変わりますが、以下にその代表的な手法を簡単に紹介します。

▼プログラム上でのバランス調整の手法

図10－1－1 手法一覧表

ランク法	イージー、ノーマル、ハード等、ユーザーに難易度のランクを選択させる事によってバランスを決める。 内部的に段階を持っており、比較的プログラムもたやすい。
レベル法	ランク法の段階をかなり細かく持ったもの。内部的に256段階など、かなり細かい調整が可能。 ただし、細かいデータを持つ必要があったり、難易度の調整などを計算式で補う必要があるためプログラムが比較的困難。
AI法	レベル法を発展させた物で、ユーザーの反応やレベルに合わせて、バランスを自動的に調整する。 もっともプログラムは面倒だが、うまく機能すれば、かなりバランスの取れる手法。
無し	アドベンチャー、RPG等。その他、アクションゲーム等で、パターン化されたものをクリアしていくタイプのゲーム。 純粋にデータのみでバランスを取るため、プログラムは単純だが、データの作成環境を充実させる必要がある。

代表的な手法だけでもこれだけの調整方法があります。

ただし、これらの方法にこだわる必要はありません。

本書の例を参考に、ゲームに合わせた調整方法を作成するのが一番でしょう。

次項からは、いくつかの例を元に、これらの方法について、実際どうプログラムしていくかを見て行きましょう。

10-2 ゲームバランスの調整－ランク法

ランク法

ランク法とは、イージーやハードなど、難易度やゲームバランスをプレイヤーに選択してもらい、調整する手法です。

厳密には調整とはいえない部分もありますが、「プレイヤーが納得するバランス＝ベストのゲームバランス」と考えれば、これも立派なバランス調整法の１つでしょう。

ランク法の調整方法

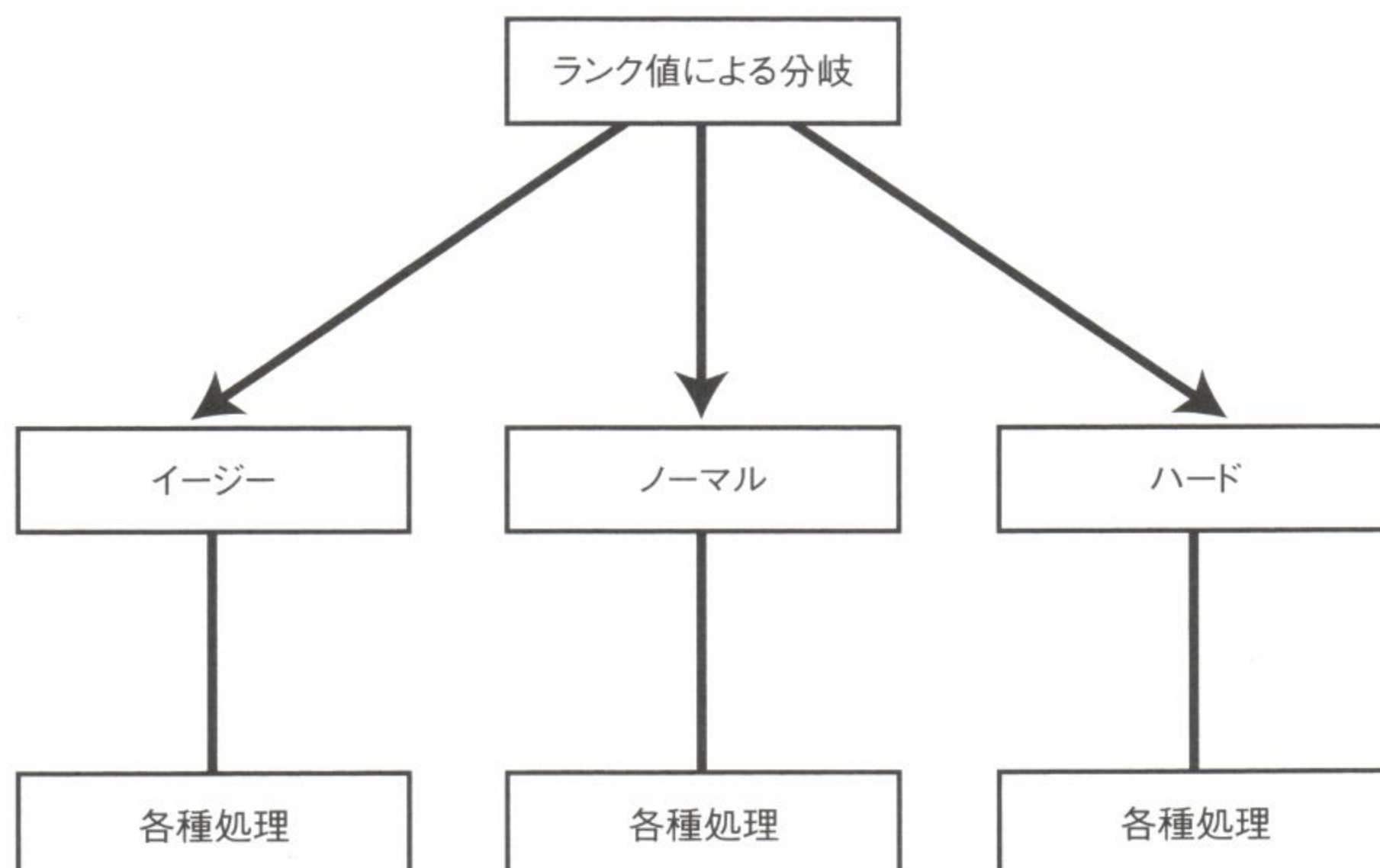
では、実際にどういう調整方法なのでしょうか。

理屈は単純で、まず、プレイヤーにゲームのランクを選択してもらいます。

選択方法は様々ですが、通常はイージー、ハード等の名称でコンフィグ画面で選択する事が多いでしょう。

その後、その選択されたランクをキーとして、バランスを決めるデータを選択します。

このデータ選択方法はゲームにより様々です。また、何処までこのキーを応用するかで、作りこみが変わってきます。

図10－2－1 ランク法のフロー図

プログラム例

同様に、実際のプログラム手法も様々ですが、一般的には以下のような形のプログラムになるでしょう。

このサンプルでは、ランクのキーを元に、弾の移動速度を変更しています。

実際には弾の速度だけではなく、弾の発射パターンや発射間隔を変えたい場合などがありますが、そのような場合は、それに準じたプログラムを用意してやる必要があります。

もちろんこれは一例で、ジャンルによって応用方法や使い方は変わります。

下記の例はそのままシューティングに応用出来ますし、アクションでは敵の出現する頻度や、相手や自分の体力を変えると良いでしょう。

LIST 10 －2－1 ゲームバランス調整 ランク法

```
#define RANK_MAX 3
#define BULLET_TIME 15
typedef struct {
    SPRITE          Sprt;
    int             Time;
    int             Rank;
} EX10_02_STRUCT;

typedef struct{
    SPRITE          Sprt;
    float           AddX;   //増分値X
    float           AddY;   //増分値Y
} EX10_02_BULLET;

void exec10_02_bullet(TCB* thisTCB)
{
    EX10_02_BULLET* work = (EX10_02_BULLET*)thisTCB->Work;

    if( work->Sprt.X < 0 || work->Sprt.X >= SCREEN_WIDTH ||
        work->Sprt.Y < 0 || work->Sprt.Y >= SCREEN_HEIGHT )
    {//画面外への移動で弾を消去
        TaskKill( thisTCB );
    }

    //座標に増分値を加算
```

```
    work->Sprt.X += work->AddX;
    work->Sprt.Y += work->AddY;

    SpriteDraw(&work->Sprt,1);
}

void init10_02(TCB* thisTCB)
{
    EX10_02_STRUCT* work = (EX10_02_STRUCT*)thisTCB->Work;

    //使用するテクスチャの読み込み
    D3DXCreateTextureFromFile( g_pD3Ddevice, "..¥¥..¥¥data¥¥0044.png",&g_pTex[0] );
    D3DXCreateTextureFromFile( g_pD3Ddevice, "..¥¥..¥¥data¥¥0055.png",&g_pTex[1] );

    //座標設定
    work->Sprt.X = SCREEN_WIDTH  / 2;
    work->Sprt.Y = SCREEN_HEIGHT / 3 * 2;
}

void exec10_02(TCB* thisTCB)
{
#define MOVE_SPEED  8.0
    EX10_02_STRUCT* work = (EX10_02_STRUCT*)thisTCB->Work;
    TCB* tmp_tcb;
    EX10_02_BULLET* tmp_work;
    char str[128];
    //目標の方向
    float direction;
    //ランクによる速度
    float rank_speed;
float speed_data[] =
{//ランク別の速度データ
    8.0,    //RANK 0(EASY)
    12.0,   //RANK 1(NORMAL)
    16.0,   //RANK 2(HARD)
};

    //キー入力による移動
    if( g_InputBuff & KEY_UP    ) work->Sprt.Y -= MOVE_SPEED;
    if( g_InputBuff & KEY_DOWN  ) work->Sprt.Y += MOVE_SPEED;
```

```
    if( g_InputBuff & KEY_RIGHT ) work->Sprt.X += MOVE_SPEED;
    if( g_InputBuff & KEY_LEFT  ) work->Sprt.X -= MOVE_SPEED;
    //ランクの切り替え
    if( g_DownInputBuff & KEY_Z ) work->Rank--;
    if( g_DownInputBuff & KEY_X ) work->Rank++;
    if( work->Rank <  0 ) work->Rank = 0;
    if( work->Rank >= RANK_MAX ) work->Rank = RANK_MAX-1;

    if(work->Time++ >= BULLET_TIME)
    {//一定時間ごとに弾を作成
        tmp_tcb = TaskMake( exec10_02_bullet, 0x2000 );
        tmp_work = (EX10_02_BULLET*)tmp_tcb->Work;

        //常に中心座標から発射
        tmp_work->Sprt.X = SCREEN_WIDTH  / 2;
        tmp_work->Sprt.Y = SCREEN_HEIGHT / 2;

        //中心座標から自機への方向を計算
        direction =
            atan2( work->Sprt.Y - tmp_work->Sprt.Y,
                   work->Sprt.X - tmp_work->Sprt.X );

        //速度はランクによって決定
        rank_speed = speed_data[ work->Rank ];

        //方向から、X、Yそれぞれの座標増分値を計算
        tmp_work->AddX = cos( direction ) * rank_speed;
        tmp_work->AddY = sin( direction ) * rank_speed;

        work->Time = 0;
    }

    SpriteDraw( &work->Sprt, 0);

    sprintf(str,"RANK = %d",work->Rank);
    FontPrint( 16,16, str);
}
```

10-3 ゲームバランスの調整－レベル法

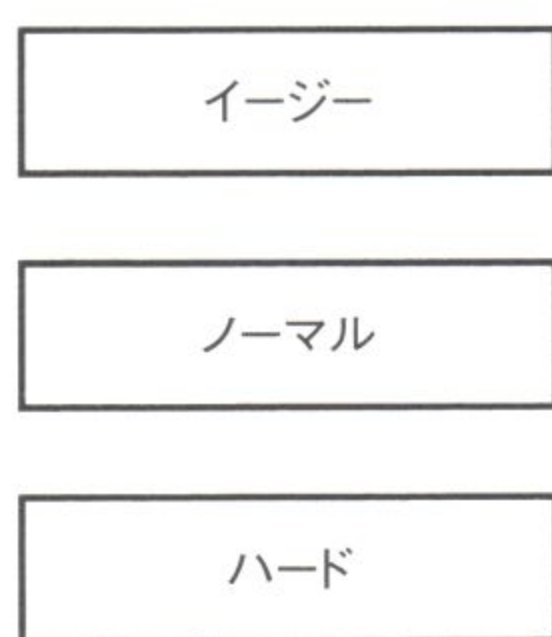
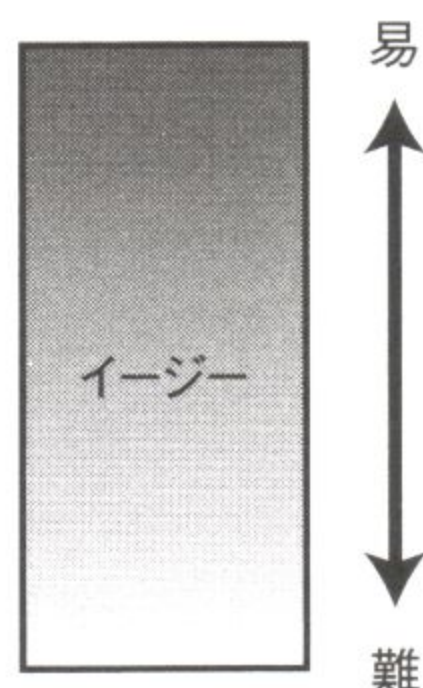
レベル法

レベル法はランク法の発展型といえます。

ランク法はプレイヤーに選択してもらうため、2～3段階、多くても5段階程度の幅でしかバランス指定が出来ませんでした。

レベル法はこのランクにおける段階を一気に緩和し、256段階や、場合によっては65536段階ものランクを持ちます。

デジタル的な段階を持つランク法と違い、アナログ的な調整が可能になり、細かい調整も可能になります。

図10－3－1 ランク法と違い、よりアナログ的な調整が可能なレベル法のイメージ図

ランク法では通常3～5段階程度

レベル法では多数の段階を持つことにより
細かな調整が可能

ただし、プレイヤーにこういったアナログ的な数値を指定してもらう事は難しいため、レベルの数値は製作側で管理する必要があります。

また、色々な調整項目において、数百～数万にも及ぶ段階のバランスデータを持つ事はかなり困難です。

そのため、データでの指定ではなく、この「レベル値」をパラメータとして、データを返す計算式をプログラムしてやる必要があります。

レベル法のプログラム

実際のプログラムです。基本的な処理の流れは、[10－2]のランク法と同じです。

サンプルでは、256段階の「レベル値」を元に、弾の速度を変えています。

一番重要な部分となる、速度のデータを返す計算式は、指定した速度の最大値と最小値の間の数値を256段階に分割して返すものです。

なお、ここでは、速度調整だけですが、実際に細かく調整を可能にするためには、いろんな項目において計算式を用意する必要があるため、どうしてもプログラムは複雑になりがちです。

そのため、実際に作成するには、「レベル値」を、ランクに変換して、単純なデータにしてしまう等の工夫も必要になります。

ただし、あまりこういったデジタル的な段階を持つデータに頼りすぎると、せっかくのアナログ的な調整が無意味なものになってしまうので、十分検討して作成して下さい。

LIST 10－3－1 ゲームバランス調整 レベル法

```
#define LEVEL_MAX 256
#define BULLET_SPEED_MIN 4.0
#define BULLET_SPEED_MAX 15.0
#define BULLET_TIME 15

typedef struct {
    SPRITE          Sprt;
    int             Time;
    int             Level;
} EX10_03_STRUCT;

float EX10_03_BulletSpeed(float Level)
{//レベル値から弾の速度を計算
    return  (BULLET_SPEED_MAX - BULLET_SPEED_MIN) * Level / LEVEL_MAX +
BULLET_SPEED_MIN;
}

void init10_03(TCB* thisTCB)
{
    EX10_03_STRUCT* work = (EX10_03_STRUCT*)thisTCB->Work;

    //使用するテクスチャの読み込み
```

```
    D3DXCreateTextureFromFile( g_pD3Ddevice, "..¥¥..¥¥data¥¥0044.png",
&g_pTex[0] );
    D3DXCreateTextureFromFile( g_pD3Ddevice, "..¥¥..¥¥data¥¥0055.png",
&g_pTex[1] );

    //座標設定
    work->Sprt.X = SCREEN_WIDTH  / 2;
    work->Sprt.Y = SCREEN_HEIGHT / 3 * 2;
}

void exec10_03(TCB* thisTCB)
{
#define MOVE_SPEED  8.0
    EX10_03_STRUCT* work = (EX10_03_STRUCT*)thisTCB->Work;
    TCB* tmp_tcb;
    EX10_02_BULLET* tmp_work;
    char str[128];
    //目標の方向
    float direction;
    //レベルによる速度
    float level_speed;


    //キー入力による移動
    if( g_InputBuff & KEY_UP    ) work->Sprt.Y -= MOVE_SPEED;
    if( g_InputBuff & KEY_DOWN  ) work->Sprt.Y += MOVE_SPEED;
    if( g_InputBuff & KEY_RIGHT ) work->Sprt.X += MOVE_SPEED;
    if( g_InputBuff & KEY_LEFT  ) work->Sprt.X -= MOVE_SPEED;
    //レベル値の増減
    if( g_InputBuff & KEY_Z ) work->Level--;
    if( g_InputBuff & KEY_X ) work->Level++;
    if( work->Level <  0 ) work->Level = 0;
    if( work->Level >= LEVEL_MAX ) work->Level = LEVEL_MAX-1;

    if(work->Time++ >= BULLET_TIME)
    {//一定時間ごとに弾を作成
        tmp_tcb = TaskMake( exec10_02_bullet, 0x2000 );
        tmp_work = (EX10_02_BULLET*)tmp_tcb->Work;

        //常に中心座標から発射
```

```
        tmp_work->Sprt.X = SCREEN_WIDTH  / 2;
        tmp_work->Sprt.Y = SCREEN_HEIGHT / 2;

        //中心座標から自機への方向を計算
        direction =
            atan2( work->Sprt.Y - tmp_work->Sprt.Y,
                   work->Sprt.X - tmp_work->Sprt.X );

        //レベル値から速度を算出
        level_speed = EX10_03_BulletSpeed(work->Level);

        //方向から、X、Yそれぞれの座標増分値を計算
        tmp_work->AddX = cos( direction ) * level_speed;
        tmp_work->AddY = sin( direction ) * level_speed;

        work->Time = 0;
    }

    SpriteDraw( &work->Sprt, 0);

    sprintf(str,"LEVEL = %3d",work->Level);
    FontPrint( 16,16, str);
}
```

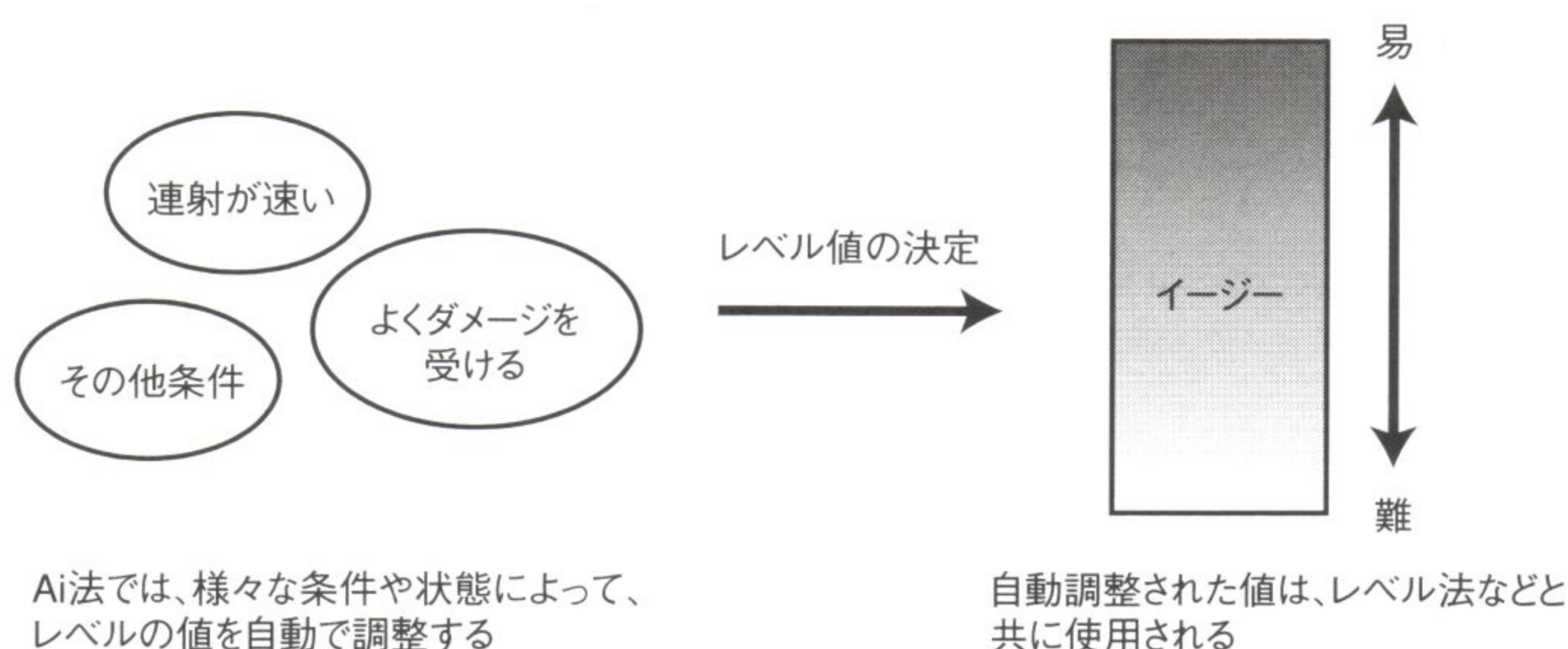
10-4 ゲームバランスの調整ー AI法

AI法

AI法は前出の2つの方法、ランク法とレベル法を組み合わせて使う事が前程の調整方法です。

前出の2つの方法は、基本的に人間が調整をしてやる必要がありました。

しかしこのAI法ではプレイヤーの行動や状況に応じて、自動的にバランス調整を行ないます。

図10ー4ー1 AI法のイメージ図

どのような行動でバランスを取るかは、ゲームによって大きく異なってしまうため、抽象的になってしまいますが、プレイヤーの腕前を見たり、プレイヤーの状況をみて判断する事が多いようです。

具体的には、以下の様なものなどでしょうか。

- **一定時間内に高得点を取ると、難易度を高めにする**
- **プレイヤーが一定回数以上の連射をしていたら、上級者とみなしてバランスを高めにする**
- **シューティングなどで残機が少なくなったら、弾の速度を遅くする**
- **アクションで体力が少なくなったら回復アイテムが出現する**

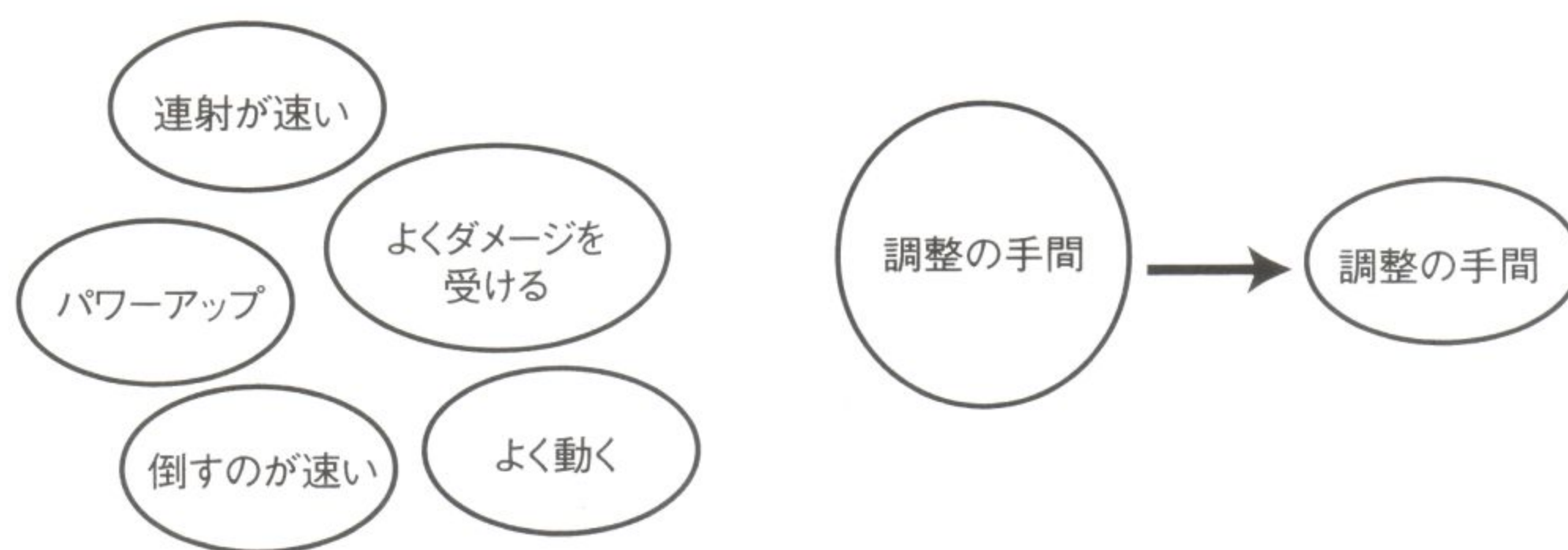
これ以外にもゲーム性と絡めて、特定の敵、例えばレーダーを集中して破壊すると難易度が下がる、等の演出としても使用されます。

細かい作りこみが出来て、演出にも使用できる、AI法ですが良い所ばかりではありません。

まず、自動的に調整とは言っても限界はありますし、実際の調整範囲の決定は、結局人の手でやらなくてはなりません。

何より、細かい調整を行なうためには様々な状況に応じた調整方法を、プログラムしなくてはならないため、作成が非常に大変です。

もっとも、自動化されることで、ゲームによっては逆に大幅に作成が楽になるケースもあるため、ゲームの内容に応じて使い分ける事が大事でしょう。

図10－4－2 AI法のメリット、デメリットのイメージ図

様々な条件や状態を監視し、それぞれについてレベル値を管理する関数やデータを作らなくてはならないため処理が非常に面倒

しかし、一旦作ったら、ある程度までは自動で調整してくれるので、調整の手間が削減される

AI法のプログラム

実際にAI法のプログラムを組む事はどうしても大規模になってしまうため、以下のプログラムは判定部分のみのプログラムです。

以下の調整条件は3つだけですが、実際には調整の数だけプログラムを書く必要があります。

```
void AI_LevelCheck( TCB* thisTCB )
{
	//プレイヤーの残機を見てLevelを調整
	g_GameLevel += PlayerZankiCheck( g_GameLevel );

	//プレイヤーのパワーアップ回数を見てLevelを調整
	g_GameLevel += PlayerPowerUPCount( g_GameLevel );

	//ライフが一定量以下でアイテムが前回出現してから一定時間が過ぎていたら緊急アイテム出現
	if( PlayerLifeCheck() <= 30 && --ItemTime <= 0 )
	{
		//1度出現したら、再出現するまでの時間を設定する
```

```
        g_ItemTime = 30000;
        //緊急用のアイテムを出現させる
        ItemAppearance( SAVE_ITEM );
    }
}
```

10-5 レベルアップしたときの数値の管理

プレイヤーが満足できるレベルアップとは

RPGなどにおいてレベルアップの方法は、システムやゲーム性と密接にかかわってきます。

そのためにはレベルアップ時のパラメータ数値を調整して、指定のレベルで指定の強さになるように管理する必要があります。

しかしこの管理が余り厳しいと、プレイヤーは「ゲームをさせられている」という、反発感を抱いてしまいます。

そのため製作者は、プレイヤーが満足するようにゆとりをもたせ、どのようにレベルアップして欲しいのかを考える必要があります。

その手法として、ここではRPGにおける、代表的なレベルアップ数値の管理法を幾つかあげてみます。

乱数法

文字通り、乱数で数値を上げる方法です。

プレイヤーにどの程度数値が上がるか予測が出来ないため、あまり良い方法ではありませんが、一喜一憂の度合いが大きいため、何度も繰り返し解くタイプのゲームには向いています。

また、運の要素が非常に大きくなるため、レベルの数値が当てにならず、同じレベルでも極端に強さが変わる場合があります。

データ法

特定の計算式や各レベルに対応したデータで数値を管理する手法です。

細かいレベル調整や、レベルアップの性格付け（最初は弱いが、後半強くなっていく等）も可能です。

反面、管理が大変で面倒な部分もあります。大規模なゲームで、細かく調整をしたいゲームに向いています。

また、数値の上がり方に対してプレイヤーの予測がつきやすいので、乱数法を折り混ぜる場合もあります。

プレイヤー設定法

数値の配分をプレイヤーに任せてしまう手法です。

管理がプレイヤーに委ねられるので、一見楽そうに思えますが、プレイヤーがどのようなパラメータにしてもクリア出来るようにする必要があります。

そのため、この手法を採用するゲームは、比較的簡単になってしまう傾向があり、ベストなゲームバランスを取るのが若干難しい手法といえます。

レベルアップのプログラム

最後にレベルアップのサンプルとして、データ型のプログラムを見ていきましょう。

まず、レベルに応じたデータを用意します。ここで用意するのは、現在のレベルから次のレベルに移る時の数値です。

そしてレベルアップ処理を行ない、データをパラメータに加算します。

ここでは、データをそのままパラメータ数値に反映していますが、状況に応じて乱数法を混ぜたり、プレイヤーが設定できる数値の下地にする等、色々加工してみてください。

最後にパラメータの表示です。レベル値は0からカウントしていますので、表示時に＋1していますが、他に大きな問題点は無いでしょう。

LIST 10－5－1 レベルアップしたときの数値の管理

```
typedef struct{
    int          Level;
    int          Str;
    int          Agi;
    int          HP;
    int          MP;
} EX10_05_STRUCT;

void exec10_05(TCB* thisTCB)
{
#define LEVEL_MAX  10

typedef struct{
    char         Str;        //力
    char         Agi;        //素早さ
```

```
    char            HP;         //HP
    char            MP;         //MP
} EX10_05_TABLE;

    EX10_05_STRUCT* work = (EX10_05_STRUCT*)thisTCB->Work;
    char str[128];
    RECT font_pos = { 0, 0, 640, 480, };
    EX10_05_TABLE LevelTable[ LEVEL_MAX ] =
{
    { 1, 5,15, 5,},  //Level 1-2
    { 2, 4, 5, 5,},  //Level 2-3
    { 3, 3, 3, 5,},  //Level 3-4
    { 4, 2,25, 5,},  //Level 4-5
    { 1, 5, 0, 5,},  //Level 5-6
    { 2, 4, 2, 5,},  //Level 6-7
    { 3, 3, 3, 5,},  //Level 7-8
    { 4, 2, 5, 5,},  //Level 8-9
    { 1, 5,12, 5,},  //Level 9-10
    { 0, 0, 0, 0,},  //Level10-11
};

    //キー入力によるレベルアップ
    if( g_DownInputBuff & KEY_Z )
    {
        //レベルが最大になっているかチェック
        if( work->Level < LEVEL_MAX - 1 )
        {//レベルアップ処理、各種パラメータ数値をアップ
            work->Str += LevelTable[ work->Level ].Str;
            work->Agi += LevelTable[ work->Level ].Agi;
            work->HP  += LevelTable[ work->Level ].HP;
            work->MP  += LevelTable[ work->Level ].MP;
            work->Level++;
        }
    }

    //レベルの初期化
    if( g_DownInputBuff & KEY_X )
    {
        work->Str = 0;
        work->Agi = 0;
```

```
            work->HP    = 0;
            work->MP    = 0;
            work->Level = 0;
        }
        //各種パラメータ表示
        sprintf( str,"レベル  %d", work->Level + 1 );
        g_pFont->DrawText( NULL, str, -1, &font_pos, DT_LEFT, 0xffffffff);
        font_pos.top += 32;

        sprintf( str,"強さ    %2d  HP %3d", work->Str,work->HP );
        g_pFont->DrawText( NULL, str, -1, &font_pos, DT_LEFT, 0xffffffff);
        font_pos.top += 16;

        sprintf( str,"素早さ  %2d  MP %3d", work->Agi,work->MP );
        g_pFont->DrawText( NULL, str, -1, &font_pos, DT_LEFT, 0xffffffff);
}
```

10-6 RPGやシミュレーションにおける敵の強さのバランス

ゲームにおける敵の強さのバランスとは

RPGやシミュレーションにおける強さのバランスを考えてみましょう。

とはいえ、一口にRPGやシミュレーションといってもいろんなタイプがありますので、ここでは限定して一般的なRPGの戦闘、いわゆるドラクエ型やウィザードリィ型を元に話を進めていきます。

ゲームの面白さ

さて、実際に敵の強さのバランスを考える場合、まずどのような戦闘が面白いのか？　敵はどのくらいの強さが理想なのか？　という事を考えなくてはなりません。

色々な考え方や意見はありますが、筆者の考えでは上記のようなゲームでの面白い戦闘とは、以下のような要素を持ちます。

1　結果にプレイヤーの努力や判断（戦略性）が反映される

これは、実際の操作に「プレイヤーの思い通りになる余地がある」という事です。

単純な話ではありますが、たとえ結果が同じでも、単にボタンを連打するだけの戦闘と、思考を繰り返して選択を行なう戦闘では後者の方が面白くなりやすいのは理解できるでしょう。

2　運によっても戦闘の結果が決まる

ここで言う運とは、プレイヤーの期待感をくすぐる事象を指します。これは、結果の良し悪しは関係ありません。

要するに「ここで、毒の攻撃を受けるかもしれない」「ここで、会心の一撃が出るかもしれない」と、良くも悪くもプレイヤーが期待をしてしまう要素です。

矛盾する2つの要素

考えてみると分かりますが、これら2つの要素は非常に矛盾した要素です。

なぜなら、完全な戦略性は運の要素を排除し、完全な運による戦闘は、戦略性の入る余地がないからです。

しかし、どちらの要素が突出しても、バランスとしては破綻してしまい、とても面白いとは思えなくなってしまいます。

例えば、意外性が無い、すなわち結果が完全に予測できる戦闘はすぐ飽きてしまいます。

また完全に運だけで結果が決まるような戦闘、例えば、何もしなくても最後のボスに勝ててしまうような戦闘では、戦闘の意味すら無くなってしまうでしょう。

要するに、これら２つの要素のバランスがこそが面白いと思える要素なのです。

実際のバランスのとり方

ではこれら２つのバランスをどうやって取り、敵の強さを決めていけばよいのでしょうか？

結論から言うと、最終的にバランスを取るには何度も繰り返し、戦闘を行ないチェックをするしかありません。

まず同じ強さにしてから調整をしていく

しかし、敵の強さに関して、ある程度の目安を置くことは出来ます。それは相手と自分のキャラクターをまったく同じステータス、すなわち同じ強さにしてしまうのです。

理論上は同じ強さなのですから、運の要素以外は確実に同じになります。しかし、敵には独特の攻撃や癖を設定するのが普通ですので、まったく同じ強さという事はありません。

そこでまずは、同じステータスを設定し、単純に同じ強さになるように調整を行います。そこから戦略性を生むような、調整を施すようにしてやります。

もう少し平たく書くと、運の要素を先に調整し、その後戦略性の調整を行なうと言った感じでしょうか。

もちろん強さは他の要素、レベルやアイテムによっても変わりますが、極端な強弱をつけない限りは十分な目安となるでしょう。

10-7 敵の出現におけるバランス

集団戦における敵の出現数と強さ

パーティー制のRPGやシミュレーションでは、パーティーやグループによる戦闘、すなわち集団戦が行なわれます。

こういった集団戦では厳密なバランスというものが難しく、圧倒的に強すぎるか、逆に圧倒に弱くなってしまいがちです。

そこで、ここでは、バランス調整の指標となる、ランチェスターの法則を紹介します。

ランチェスターの法則とは、イギリスの技術者ランチェスターが発見した法則です。

2つの法則があり、どちらも集団における戦闘の基本的な考え方を示した物で、集団内の各個人が同じ能力を持っている事を前程としています。

ランチェスターの第一法則

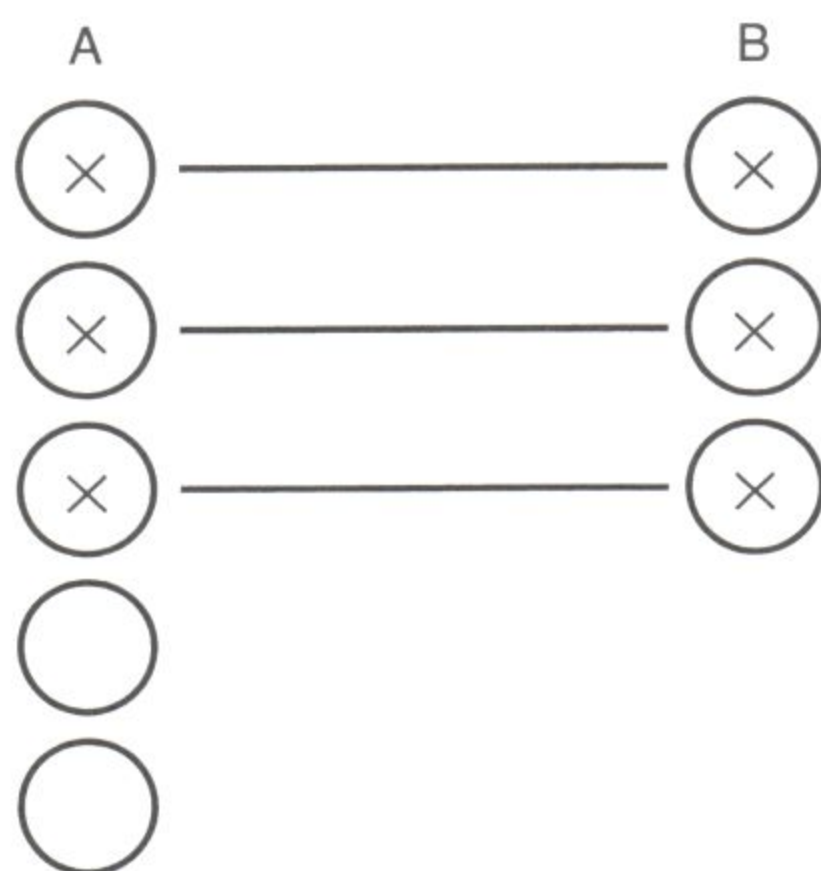
「集団戦において、1対1でしか戦えないのであれば、数の多い方がその差だけ勝つ」

この法則はは一騎打ちの法則とも呼ばれ、単純に各集団が、1対1で戦った場合は数の多い方が勝つというものです。

各個人が同じ能力を持っている場合、同じ強さ、同じ数の集団と戦った場合、全滅する可能性があるという事を示しています。

もしこの法則を元にバランスを考えるのであれば、単純にプレイヤー側より少し少ない数を出すようにするとよいでしょう。

図10 ―8―1 ランチェスターの第一法則のイメージ図

A B

1対1で戦った場合、Aの方が2人分の戦力を残して勝つ

ランチェスターの第二法則

「集団戦において、１人が複数の相手に対して攻撃できるのであれば、その差は２乗に比例する」

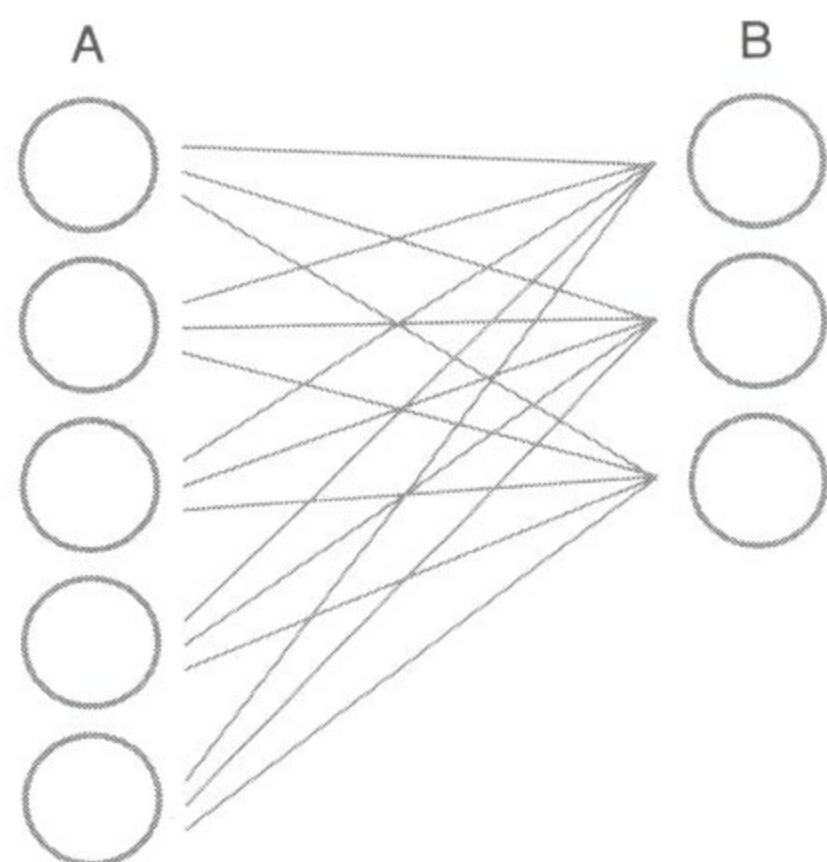
第二法則は「２乗の法則」とも呼ばれ、第１法則よりも少し複雑です。

もう少し細かく書くと、３人対９人で戦闘を行なった場合、その戦力は３倍ではなく９倍になるということです。

図を見てもらえれば気がつくと思いますが、第二法則における戦力差は単純な倍数ではなく、ずっと大きな差（二乗数）である事がわかります。

もしこの法則を元にバランスを取るのであれば、わずかな数の差が致命的にならないようにレベル制など、個々の質を高めるようなシステムを導入するべきでしょう。

図10 ―8―2 ランチェスターの第二法則のイメージ図

A B

複数同時に攻撃できる場合、その戦力差は2乗に比例する

この法則を念頭に入れて、調整やデータの設定を行なえば、酷いゲームバランスという事にはなりにくいでしょう。

もちろん、このランチェスターの法則は絶対的な法則ではありません。

RPGにはレベルやアイテム等の「質」を高める要素がありますし、シミュレーションでは、地形などの要素が加わる事もあります。

あくまでこの法則は、基本となる考えと目安を提示するだけです。

どのような手法であっても一番ベストな方法は、実際にゲームをプレイしながら調整を繰り返す事である事を、覚えておいてください。

Chapter 11

サウンド

11-1 音をならすプログラミングの基本

BGMと効果音

ゲームで使用される音は大きく分けて2つあります。BGMと効果音です。

再生そのものに基本的な違いはありませんが、BGMは容量が大きくなりやすいので、特殊な処理が必要になります。

逆にSEは、容量はさほど大きくなく、またゲーム上で頻繁に繰り返し呼ばれるため、ゲーム開始時にあらかじめメモリ上に読み込んでおく事が多くなっています。

また、リアルタイムのゲームでは、画面と音がずれてしまうとかなりの違和感が生じるため再生の反応性も重要になってきます。

以上の差があるため、BGMと効果音の再生は別々に分けて処理する事が一般的です。

図11－1－1 BGMと効果音の違い 再生手法の違い

BGMの特徴

再生時間が長い
データ容量が大きい
同時に再生される数は1～2個
比較的高音質が求められる
再生時に特殊な処理が必要

SEの特徴

再生時間が短い
1つ当たりのデータ容量が小さい
多数のデータが同時に再生される
即時再生が求められる
メモリ上にあるデータの再生が手軽

使用するAPI

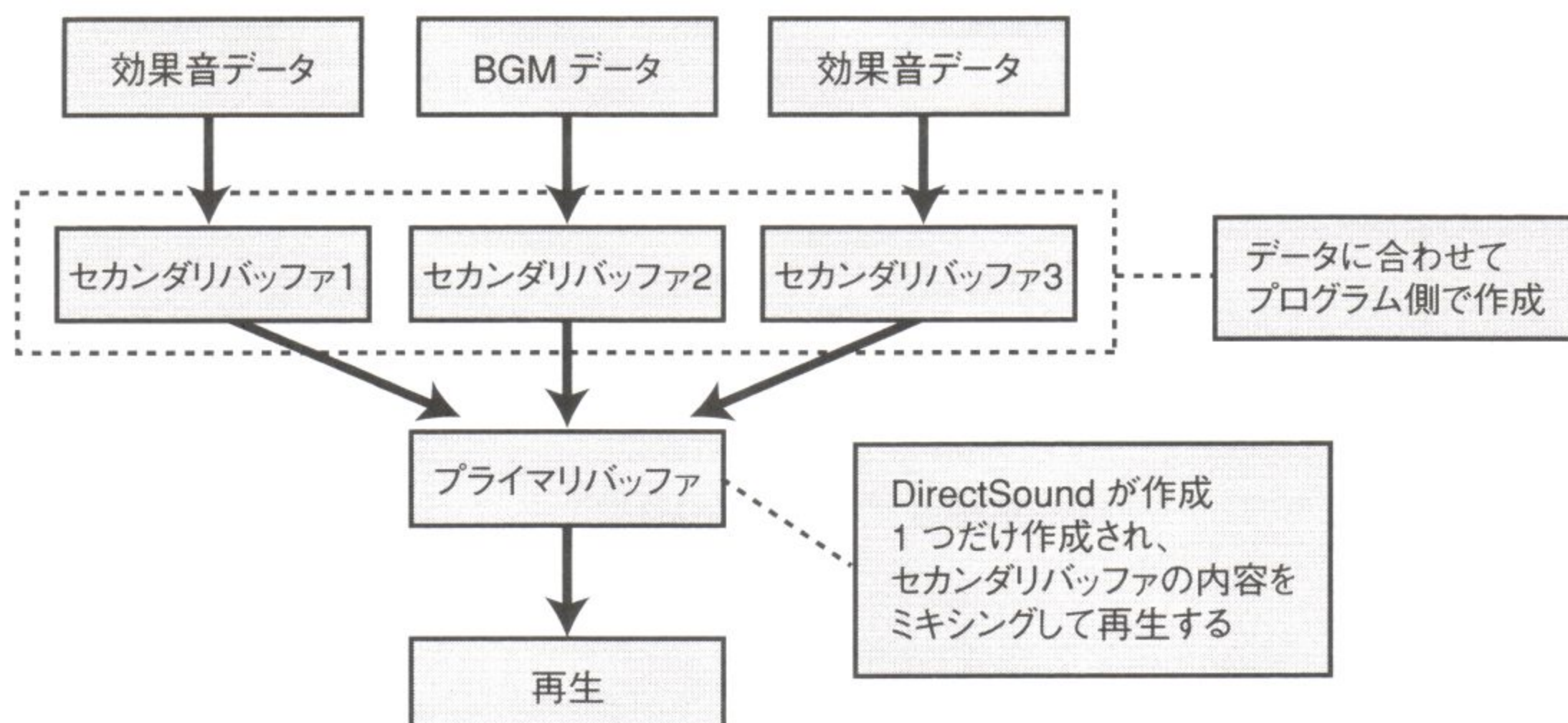
使用するAPIですが、これも何種類かあります。本書ではゲーム用ということもあり、DirectXのサウンド機能であるDirectSoundAPIを使用します。

このAPIは、他の再生APIと比較して反応速度が速く、ゲーム画面と音がずれるといった現象が起りにくくなっています。

DirectSoundの概念

ここでは、以降の項目の前準備の意味も含めて再生APIであるDirectSoundの概念と初期化処理を見ていきます。

まず、次の図を見てください、これはDirectSoundの概念図です。

図 11 ー１ー２ DirectSound とサウンドバッファ

サウンドを再生する場合、再生データはセカンダリバッファと呼ばれるバッファに格納する必要があります。

セカンダリバッファは再生データを保持しておくためのバッファで、再生データに対して１つ以上用意され、プログラマが作成する必要があります。

このセカンダリバッファの内容は、再生時にプライマリバッファと呼ばれる再生用のバッファに転送されて再生されます。

プライマリバッファはセカンダリバッファとは違い、DirectSound の初期化時に自動的に作成され、プログラマが作成する必要はありません。

また、複数のデータを同時に再生した場合、このプライマリバッファでミキシングされてから再生されます。

DirectSound の初期化処理

次に初期化処理を見てみましょう。

とは言うものの、ほとんどする事はなく、DirectSound のオブジェクトを作成する関数を呼ぶだけでほぼ終了します。

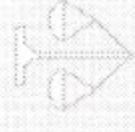
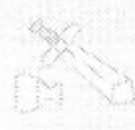
作成する関数は、DirectSoundCreate8 です。引数として、オブジェクトを格納するポインタを渡します。

```
//DirectSoundオブジェクト作成
if(FAILED(DirectSoundCreate8( NULL, &g_pDSound ,NULL ) ))
{
```

```
    return(false);
}

//協調レベルの設定
g_pDSound->SetCooperativeLevel( g_hWnd, DSSCL_PRIORITY );
```

オブジェクトを作成した後は、他に再生されるサウンドとの協調レベルをSetCooperativeLevelを呼び出して設定します。

この設定を忘れると、サウンドを再生することが出来ません。

以上で、DirectSound再生の準備は終了です。

11-2 単純再生

WAVファイルの再生

音声再生の基本である、WAVファイルの単純な再生を行なってみましょう。

再生のためには、まず再生するWAVファイルを読み込まなくてはいけません。

ところが、DirectSoundには、WAVファイルを直接読み込むAPIが用意されていないので、自分でWAVファイルの構造を解析して読み込む必要があります。

ただこれは、読み込むWAVファイルの種類を限定すればさほど難しい事はありません。

WAVファイルを読み込む関数をつくる

限定したWAVファイルを読み込むための関数ReadWaveFileを作成してみます。

この関数は非圧縮PCM、単一データのWAVファイルのみを扱い、WAVEデータ再生に必要なフォーマット情報も同時に取得します。

図11−2−1 WAVファイルの構造

4Byte	RIFFファイルである事を示す文字列'RIFF'
4Byte	先頭8Byteを除いたファイルサイズ
4Byte	WAV形式である事を示す文字列'WAVE'
4Byte	フォーマットチャンクである事を示す文字列'fmt'
4Byte	フォーマットチャンクのサイズ リニアPCMなら通常は16
16Byte	WAVEデータに関するフォーマット情報 リニアPCMなら通常16Byte固定だが保証は無い
4Byte	データチャンクである事を示す文字列'data'
4Byte	WAVEデータのサイズ
nByte	実際のWAVEデータ

WAVファイルを開く

まずは読み込むWAVファイルを開きます。開く際にはテキストモードではなく、バイナリモードで開いている事に注意してください。

ヘッダ情報を読み飛ばす

開いたあと、チャンクと呼ばれるヘッダ情報を読み飛ばします。

本来、ここで細かなチェックが必要なのですが、読み込むデータを決めてしまう事でプログラムを単純化しています。

```
//チャンク文字"RIFF"と、RIFFデータサイズを読み飛ばす
fread( dummy, 1, 4 + 4, fp );
```

フォーマット情報を読む

チャンク文字 "fmt " まで読み飛ばしたら、再生に必要なフォーマット情報を読み取ります。

フォーマット情報とは、サンプリング情報や、再生周波数など、WAVE データフォーマット固有の情報の事です。

```
//フォーマット情報取得　リニアPCMデータのみなので、拡張情報は存在しない
fread( pWaveFormat, sizeof( WAVEFORMATEX ) - 2, 1, fp );
```

サイズを取得する

次に、WAVE データのサイズを取得します。

データのサイズはチャンク文字、「data」の後に書き込まれています。

ただし、「data」の位置は決まっていないので、検索してやる必要があります。

```
//チャンク文字"data"を読み飛ばす
ZeroMemory(dummy, sizeof( dummy ));
while( strcmp( "data", dummy ) )
{
    fread( dummy, 1, 1, fp );
    if( dummy[0] == 'd' ) fread( &dummy[1], 1, 3, fp );
}
//Waveデータサイズ取得
fread( &wave_size, sizeof( int ), 1, fp );
```

バッファに読み込む

最後に、WAVE データを指定されたバッファに読み込みます。

```
//waveデータ取得
fread( pWaveData, 1, wave_size, fp );
```

以上で、WAV ファイルの読み込み関数の作成は終了です。

読み込んだWAVを再生する

次に読み込んだ情報を元にサウンドを再生してみましょう。

セカンダリバッファ作成の準備

まずは、セカンダリバッファを作成し、バッファにWAVEデータを転送します。

セカンダリバッファ作成のための準備として、DSBUFFERDESC構造体に、読み込んだデータの情報を設定します。

```
//取得した情報を設定
g_WaveControllBGM->DSBDesc.dwSize          = sizeof( DSBUFFERDESC );
g_WaveControllBGM->DSBDesc.dwFlags         = DSBCAPS_LOCSOFTWARE;
g_WaveControllBGM->DSBDesc.dwBufferBytes   = wave_size;
g_WaveControllBGM->DSBDesc.lpwfxFormat     = &g_WaveControllBGM->WaveFormat;
```

設定する内容は上から順番に、DSBUFFERDESCのサイズ、再生についてのフラグ、WAVEデータのバイト単位での容量、読み込んだフォーマット情報が格納されている構造体へのポインタです。

難しい所はありませんが、再生についてのフラグは再生時だけではなく、ここでも指定する事に留意してください。

セカンダリバッファの作成

情報の設定後、DIRECTSOUND8インターフェースのCreateSoundBufferメソッドを呼び出して、セカンダリバッファを作成します。

1つ目の引数は先ほど設定したDSBUFFERDESC構造体へのポインタ、2つめは作成されたセカンダリバッファの場所を受け取るポインタへのポインタです。

```
//サウンドバッファ作成
g_pDSound->CreateSoundBuffer( &g_WaveControllBGM->DSBDesc, &g_WaveControllBGM->pDSBuffer, NULL );
```

WAVEデータの転送

作成後、WAVEデータを転送するために、Lockメソッドを使い、バッファをロックします。

ロックとは、サウンドバッファを書き換える際に、書き換えるポイントを他のプログラムからアクセスしないように、LOCK(固定)する事です。

その後、memcpy関数を使い、読み込んだWAVEデータをサウンドバッファにコピーします。

最後にUnLockメソッドを使い、ロックしたバッファを解放します。

以上で再生の準備は終了です。

再生する

後は再生をするだけですが、再生の準備に比べて再生はとても簡単です。

サウンドバッファのインターフェースにある、Playメソッドを呼び出すだけで再生処理が開始されます。

```
    //再生
    if( g_DownInputBuff & KEY_Z ) g_WaveControllBGM->pDSBuffer->Play( 0, 0, 0 );
```

LIST 11 －2－1 単純再生

```
////////////////////////////////////////////////////////////////////
//ReadWaveFile: WAVファイルの情報取得と読み込みを行う
//              リニアPCMのみ対応、tagも単一のdataのみ対応
//
//char *pFileName:              ファイル名
//WAVEFORMATEX* pWaveFormat:    情報を格納する構造体へのポインタ
//char* pWaveData:              読み込まれるWAVデータへのポインタ
////////////////////////////////////////////////////////////////////
int ReadWaveFile( char *pFileName, WAVEFORMATEX* pWaveFormat, char* pWaveData )
{
    FILE *fp;
    int wave_size;
    char dummy[16];

    fp = fopen( pFileName, "rb" );
    if ( !fp ) return( false );

    //チャンク文字"RIFF"と、RIFFデータサイズを読み飛ばす
    fread( dummy, 1, 4 + 4, fp );

    //チャンク文字"WAVE"と、"fmt "を読み飛ばす
    fread( dummy, 1, 4 + 4, fp );

    //fmtデータサイズエリアを読み飛ばす
    fread( dummy, sizeof( int ), 1, fp );
```

```
    //フォーマット情報取得 リニアPCMデータのみで拡張情報は存在しない
    fread( pWaveFormat, sizeof( WAVEFORMATEX ) - 2, 1, fp );

    //チャンク文字"data"を読み飛ばす
    ZeroMemory(dummy, sizeof( dummy ));
    while( strcmp( "data", dummy ) )
    {
        fread( dummy, 1, 1, fp );
        if( dummy[0] == 'd' ) fread( &dummy[1], 1, 3, fp );
    }

    //Waveデータサイズ取得
    fread( &wave_size, sizeof( int ), 1, fp );

    //waveデータ取得
    fread( pWaveData, 1, wave_size, fp );

    fclose( fp );

    return( wave_size );
}
////////////////////////////////////////////////////
//11章－2 単純再生
////////////////////////////////////////////////////

void init11_02(TCB* thisTCB)
{
    int wave_size;
    //アクセス可能なバッファのサイズ
    DWORD  buff_size1, buff_size2;
    //WAVバッファアクセスポイントを格納する為のポインタ
    LPVOID pvAudioPtr1, pvAudioPtr2;

    //Waveファイルの情報とデータ取得
    wave_size = ReadWaveFile( "..¥¥..¥¥data¥¥bgm00.wav", &g_WaveControllBGM-
>WaveFormat, g_WaveControllBGM->WaveData);

    //取得した情報を設定
    g_WaveControllBGM->DSBDesc.dwSize          = sizeof( DSBUFFERDESC );
```

```
    g_WaveControllBGM->DSBDesc.dwFlags        = DSBCAPS_LOCSOFTWARE;
    g_WaveControllBGM->DSBDesc.dwBufferBytes = wave_size;
    g_WaveControllBGM->DSBDesc.lpwfxFormat    = &g_WaveControllBGM-
>WaveFormat;

    //サウンドバッファ作成
    g_pDSound->CreateSoundBuffer(&g_WaveControllBGM->DSBDesc,
                                 &g_WaveControllBGM->pDSBuffer, NULL );

    //バッファロック
    g_WaveControllBGM->pDSBuffer->Lock( 0, wave_size, //WAVEデータサイズ分だけロックする
                                        &pvAudioPtr1, &buff_size1,
                                        &pvAudioPtr2, &buff_size2,
                                        //全ブロックをロック
                                        DSBLOCK_ENTIREBUFFER
                                       );

    //サウンドデータをバッファへ書き込む
    memcpy( pvAudioPtr1, g_WaveControllBGM->WaveData, wave_size );

    //ロック解除
    g_WaveControllBGM->pDSBuffer->Unlock(
            pvAudioPtr1, buff_size1,
            pvAudioPtr2, buff_size2 );

}

void exec11_02(TCB* thisTCB)
{
    //再生
    if( g_DownInputBuff & KEY_Z )
        g_WaveControllBGM->pDSBuffer->Play( 0, 0, 0 );
    //停止
    if( g_DownInputBuff & KEY_X )
        g_WaveControllBGM->pDSBuffer->Stop();

}
```

11-3 フェードイン／アウト

サウンドのフェードイン／アウト

サウンド再生のフェードイン／アウトを行なってみましょう。

フェードイン／アウトは、曲の再生時に音量を調整する事で行ないます。

初期設定

まずは、初期設定です。WAVE データを読み込み、再生方法を設定します。

基本のポイントは単純な再生と同じですが、フラグについては気をつけてください。ここで再生のフラグを設定しておかないと、音量を変更する事が出来ません。

また、他の設定（ステレオの変更等）もここで設定しておかないと、変更が出来ない事があるため注意が必要です。

```
    g_WaveControllBGM->DSBDesc.dwFlags         =
DSBCAPS_LOCSOFTWARE|DSBCAPS_CTRLVOLUME;   //ボリュームが変えられるように指定
```

フェードイン／アウトはフェードのカウントに合わせたボリューム（音量）を、毎フレーム設定する事で行ないます。

ボリュームの設定はデシベルという単位で行なわれますが、このプログラムでは、コードの簡略化のため、フェード値＝ボリューム値としています。

フェードイン／アウトのプログラミング

さて、実際のコードです。

フェード中は、現在のフェードの値を維持する必要があるため、タスク上にフェード値を保持しています。

フェードを開始するとイン／アウト、それぞれの状態にあわせてフェード状態に入ります。

フェード状態の間は、常にフェード値をカウント、監視しており、フェード値が最小または最大値になった時にフェードを終了します。

あとは、毎フレーム、フェード値をボリューム値として設定して下さい。

LIST 11－3－1 フェードイン/アウト

```
void init11_03(TCB* thisTCB)
{
    int wave_size;
    //アクセス可能なバッファのサイズ
    DWORD  buff_size1, buff_size2;
    //WAVバッファアクセスポイントを格納する為のポインタ
    LPVOID pvAudioPtr1, pvAudioPtr2;

    //Waveファイルの情報とデータ取得
    wave_size = ReadWaveFile( "..¥¥..¥¥data¥¥bgm00.wav", &g_WaveControllBGM-
>WaveFormat, g_WaveControllBGM->WaveData);

    //取得した情報を設定
    g_WaveControllBGM->DSBDesc.dwSize        = sizeof( DSBUFFERDESC );
    g_WaveControllBGM->DSBDesc.dwFlags       =
DSBCAPS_LOCSOFTWARE|DSBCAPS_CTRLVOLUME;  //ボリュームが変えられるように指定
    g_WaveControllBGM->DSBDesc.dwBufferBytes = wave_size;
    g_WaveControllBGM->DSBDesc.lpwfxFormat   = &g_WaveControllBGM-
>WaveFormat;

    //サウンドバッファ作成
    g_pDSound->CreateSoundBuffer( &g_WaveControllBGM->DSBDesc,
                                  &g_WaveControllBGM->pDSBuffer, NULL );

    //バッファロック
    g_WaveControllBGM->pDSBuffer->Lock( 0, wave_size, //WAVEデータサイズ分だけロックする
                                        &pvAudioPtr1, &buff_size1,
                                        &pvAudioPtr2, &buff_size2,
                                        //全ブロックをロック
                                        DSBLOCK_ENTIREBUFFER
                                       );

    //サウンドデータをバッファへ書き込む
    memcpy( pvAudioPtr1, g_WaveControllBGM->WaveData, wave_size );

    //ロック解除
    g_WaveControllBGM->pDSBuffer->Unlock( pvAudioPtr1, buff_size1,
                                          pvAudioPtr2, buff_size2 );
```

```
}

void exec11_03(TCB* thisTCB)
{
#define FADE_IN  1
#define FADE_OUT 2
#define FADE_STEP (DSBVOLUME_MIN / 100)

    int* fade_count = thisTCB->Work;
    int* fade_mode  = &thisTCB->Work[1];

    if( *fade_mode == 0)
    {
        //再生
        if( g_DownInputBuff & KEY_Z )
            g_WaveControllBGM->pDSBuffer->Play( 0, 0, DSBPLAY_LOOPING );
        //停止
        if( g_DownInputBuff & KEY_X )
            g_WaveControllBGM->pDSBuffer->Stop();

        //フェード開始のチェック
        if( g_DownInputBuff & KEY_UP    )
        {   //フェードイン開始
            *fade_count = DSBVOLUME_MIN;
            *fade_mode = FADE_IN;
        }
        if( g_DownInputBuff & KEY_DOWN )
        {   //フェードアウト開始
            *fade_count = DSBVOLUME_MAX;
            *fade_mode = FADE_OUT;
        }

    }else if( *fade_mode == FADE_IN)
    {//フェードイン処理
        *fade_count -= FADE_STEP;
        if( *fade_count >= DSBVOLUME_MAX)
        {
            *fade_count = DSBVOLUME_MAX;
            *fade_mode = 0;
        }
```

```
        g_WaveControllBGM->pDSBuffer->SetVolume( *fade_count );
    }else if( *fade_mode == FADE_OUT)
    {//フェードアウト処理
        *fade_count += FADE_STEP;
        if( *fade_count <= DSBVOLUME_MIN)
        {
            *fade_count = DSBVOLUME_MIN;
            *fade_mode = 0;
        }

        g_WaveControllBGM->pDSBuffer->SetVolume( *fade_count );
    }

}
```

11-4 サウンドを途切れたところから再生する

任意のポイントから再生する

再生しているサウンドを一旦停止させるには、サウンドバッファのインターフェースにある、Stopメソッドを呼び出します。

停止されたサウンドは、再度Playを呼び出す事で、停止したポイントから再生されます。

ただしこの方法では、最初から再生したり、停止した時間に合わせて再生したりなど、任意のポイントから再生することは出来ません。

曲を好きなポイントから再生させるには、SetCurrentPositionを使います。

このメソッドの引数に、再生するポイントを指定してやれば、再生時に指定されたポイントから再生されます。

以下の例では、再生時に曲の始めから再生されます。

```
g_WaveControllBGM->pDSBuffer->SetCurrentPosition( 0 );
g_WaveControllBGM->pDSBuffer->Play( 0, 0, 0 );
```

11-5 効果音再生

ゲームを作成する上で効果音は必須と言ってもいいでしょう。無音で寂しいゲームでも、効果音があるだけで一気にゲームらしくなります。

効果音再生のプログラミング

まず、効果音を再生するためのセカンダリバッファを作成します。セカンダリバッファは、効果音の数だけ必要となりますので通常は配列構造等で管理します。

```
EXTERN WAVE_CONTROLL_STRUCT*    g_WaveControllSE[ SE_MAX ];            //SE管理用
```

その後の初期化、バッファの作成の手順は、単純な再生の時とほぼ同一です。

実際の再生も、ほぼ一緒で、以下の例では入力キーの上下左右にあわせて、4種類の効果音を再生しています。

効果音再生時の注意点

ただし、注意しなくてならないのは、同じ効果音の再生中に再度Playメソッドを呼び出しても、効果音は最初から再生されない事です。

そのため、SetCurrentPositionを呼び出し、再生のたびに、再生ポイントを一番始めに設定しています。

LIST 11－5－1 SE再生

```
void init11_05(TCB* thisTCB)
{
    int loop;
    int wave_size;
    //アクセス可能なバッファのサイズ
    DWORD  buff_size1, buff_size2;
    //WAVバッファアクセスポイントを格納する為のポインタ
    LPVOID pvAudioPtr1, pvAudioPtr2;

char* se_file[] =
```

```
{
    "..¥¥..¥¥data¥¥se00.wav",
    "..¥¥..¥¥data¥¥se01.wav",
    "..¥¥..¥¥data¥¥se02.wav",
    "..¥¥..¥¥data¥¥se03.wav",
};

    for( loop = 0; loop < 4 ; loop++ )
    {
        //Waveファイルの情報とデータ取得
        wave_size =
        ReadWaveFile( se_file[ loop ],
            &g_WaveControllSE[ loop ]->WaveFormat,
            g_WaveControllSE[ loop ]->WaveData);

        //取得した情報を設定
        g_WaveControllSE[ loop ]->DSBDesc.dwSize        = sizeof( DSBUFFERDESC );
        g_WaveControllSE[ loop ]->DSBDesc.dwFlags       = DSBCAPS_LOCSOFTWARE;
        g_WaveControllSE[ loop ]->DSBDesc.dwBufferBytes = wave_size;
        g_WaveControllSE[ loop ]->DSBDesc.lpwfxFormat   = &g_WaveControllSE[
loop ]->WaveFormat;

        //サウンドバッファ作成
        g_pDSound->CreateSoundBuffer( &g_WaveControllSE[ loop ]->DSBDesc,
                                      &g_WaveControllSE[ loop ]->pDSBuffer,
                                      NULL );

        //バッファロック
        //WAVEデータサイズ分だけロックする
        g_WaveControllSE[ loop ]->pDSBuffer->Lock( 0, wave_size,
                                                   &pvAudioPtr1, &buff_size1,
                                                   &pvAudioPtr2, &buff_size2,
                                                   //全ブロックをロック
                                                   DSBLOCK_ENTIREBUFFER
                                                  );
        //サウンドデータをバッファへ書き込む
        memcpy( pvAudioPtr1, g_WaveControllSE[ loop ]->WaveData, wave_size );

        //ロック解除
        g_WaveControllSE[ loop ]->pDSBuffer->Unlock( pvAudioPtr1, buff_size1,
```

```
                                        pvAudioPtr2, buff_size2 );
    }

}

void exec11_05(TCB* thisTCB)
{
    int loop;
    int key_data[] = { KEY_UP, KEY_DOWN, KEY_LEFT, KEY_RIGHT };

    for( loop = 0; loop < 4 ; loop++ )
    {
        //入力キーに合わせてＳＥを再生
        if( g_DownInputBuff & key_data[ loop ] )
        {
            //SEなので常に最初から再生
            g_WaveControllSE[ loop ]->pDSBuffer->SetCurrentPosition( 0 );
            g_WaveControllSE[ loop ]->pDSBuffer->Play( 0, 0, 0 );
        }
    }
}
```

11-6 BGMのストリーミング再生

ストリーミング再生とは

BGMで必要とされるデータは、短い曲や効果音などと違い、容量が非常に大きくなります。

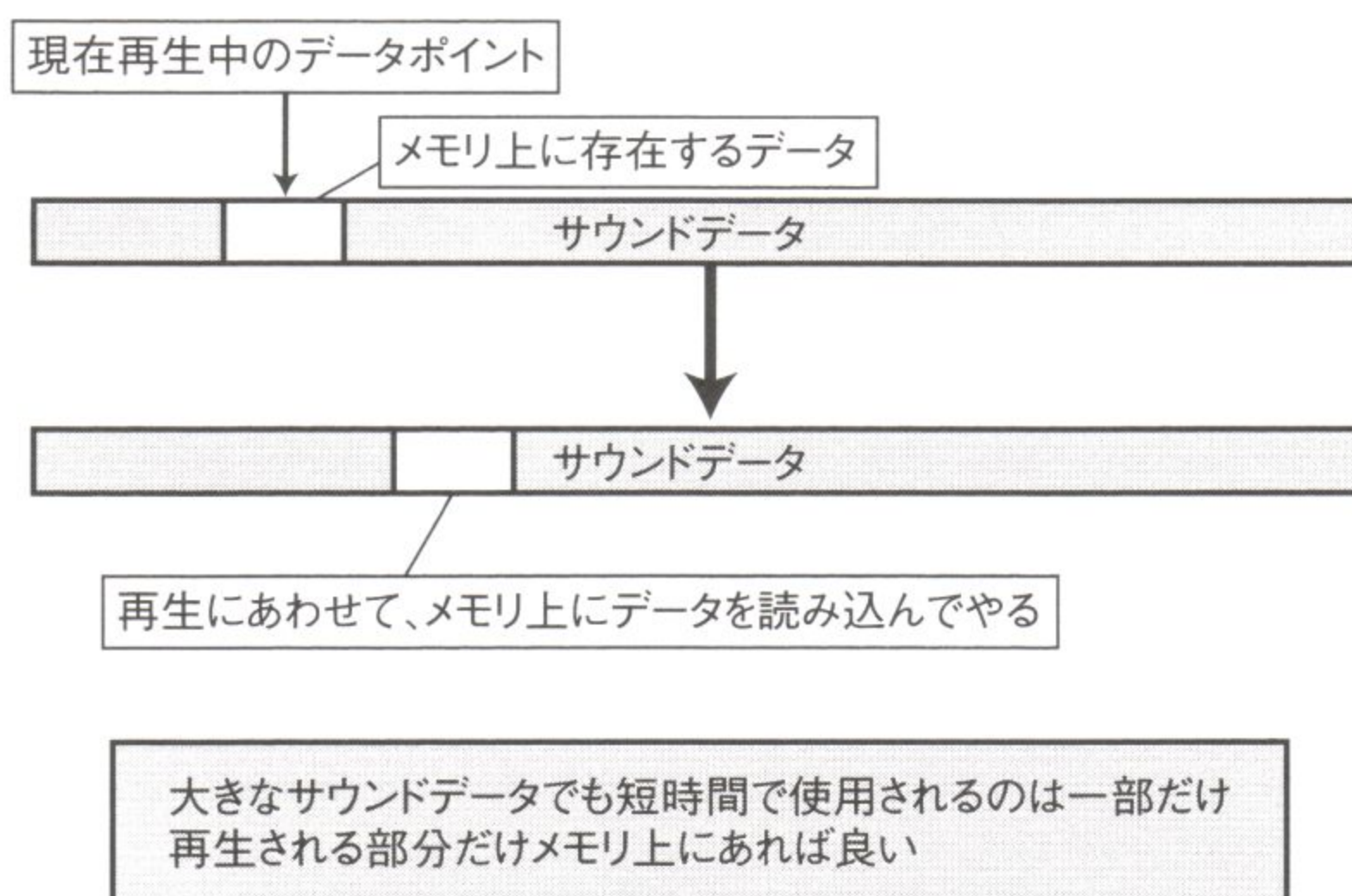
ケースによりますが、おおよそ数MByte～数十MByte、場合によっては100MByteを超えてしまう事も珍しくありません。

これだけの容量のデータを、一度にメモリに読み込むのは現実的ではありません。

そこで、通常BGMの再生にはストリーミング再生という手法を使います。

ストリーミング再生とは一方的な流れを持つデータを、少量のメモリで処理、再生する手法です。

一方的な流れを持つデータは再生するポイントのデータだけ保持していればよいことが多く、次々とデータを読み込み再生していくことで、少量のメモリでも再生することが出来ます。

図11－6－1 ストリーミング再生のイメージ図

ストリーミング再生の方法

再生データの任意の部分を読み込む関数をつくる

さて、実際にストリーミング再生をするには、データの任意の部分を任意の容量だけ読み込む必要があります。

そのため、再生データの任意の部分を読み込む関数ReadWaveFileOffsetを作成しています。

この関数は、機能としてはReadWaveFileとほぼ同じですが、指定したオフセット値から、任意のサイズのデータを読み出せる点が違います。

任意の場所を読み込める訳

DirectSoundでは、バッファのロック時にカーソルと呼ばれる、書き換えが可能なバッファの位置を示すポインタが取得できます。

データが一定量以上再生された時に、このカーソルの位置に次に再生されるデータを読み込んでやります。

そしてそのデータも再生が終わったら、再度カーソルを取得して、次に再生されるデータを読み込みます。

こうして次々に、再生される分のデータを読み込んでやれば、ストリーミング再生は実現できます。

実際のプログラミング

では実際のコードを見てみましょう。

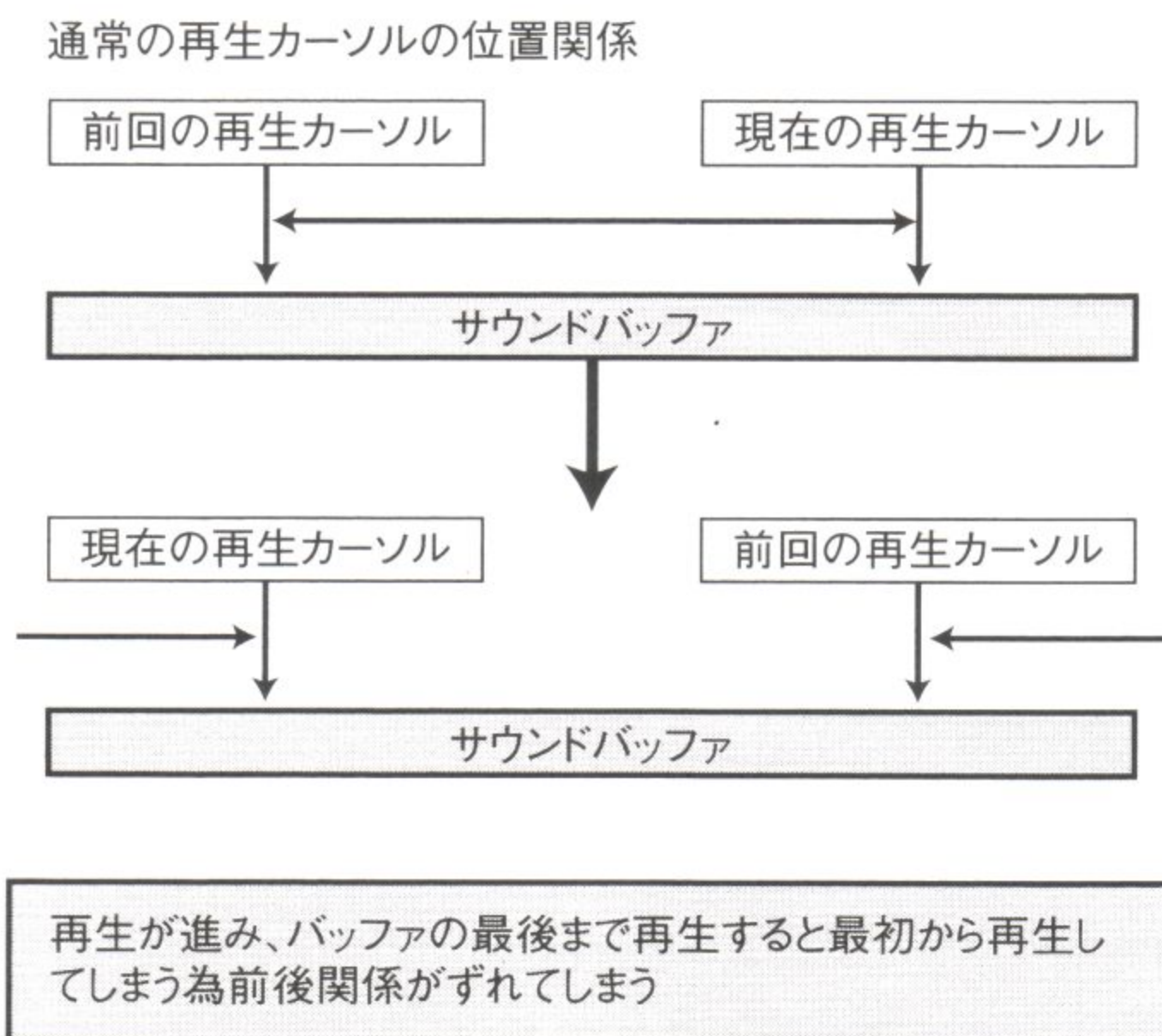
再生中のカーソル位置を取得

```
    //現在のカーソルを取得
    g_WaveControllBGM->pDSBuffer->GetCurrentPosition( NULL, &CurrentWriteCursor );

    //再生済みの容量を取得
    play_size = CurrentWriteCursor - g_WaveControllBGM->BeforePlayCursor;

    if( play_size < 0 )
    {
        play_size = (CurrentWriteCursor + BGM_READ_SIZE * 3) - g_WaveControllBGM-
>BeforePlayCursor;
    }
```

まず、GetCurrentPositionで現在再生中のカーソル位置を取得します。

取得後、前回読み込んだカーソル位置から、再生済みのサイズを示すplay_sizeを計算します。

この際、play_sizeが負の値になっていたら、カーソルの前後関係が逆になってしまっているので、再生済みサイズを再計算します。

図11－6－2 カーソルの前後関係がずれるケースのイメージ図

次のデータを読み込む処理

次に、再生された容量が一定量を超えているかをチェックし、超えていればデータを読み込み、セカンダリバッファを更新します。

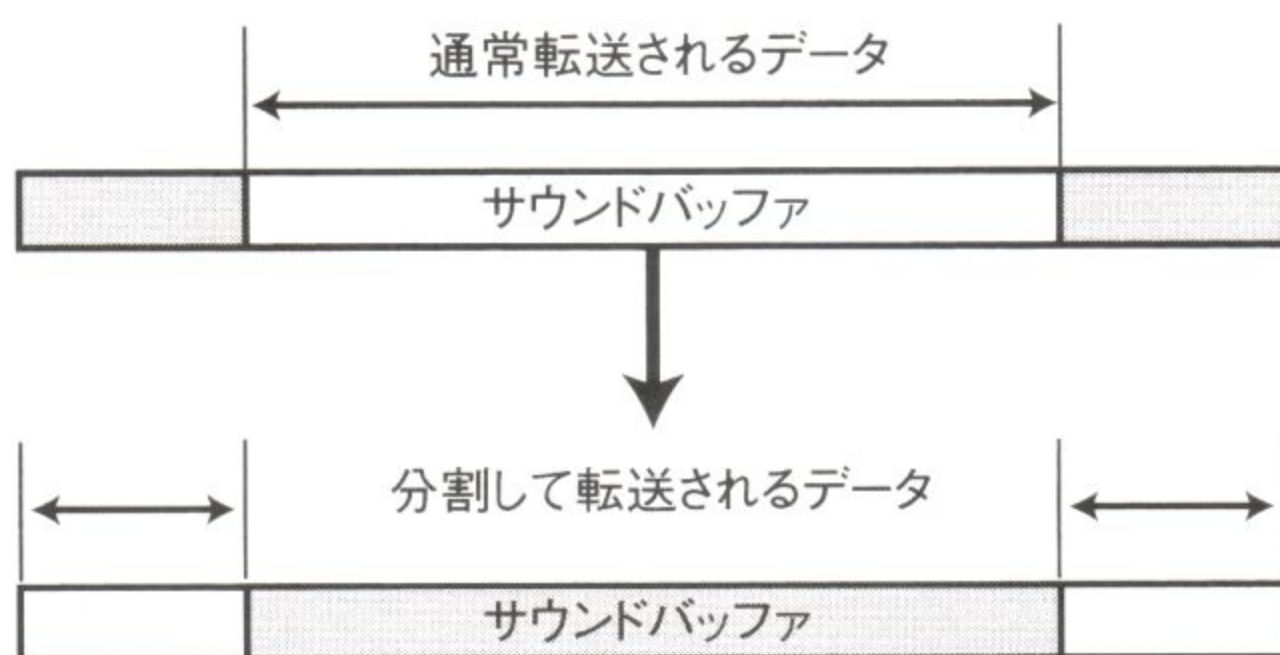
更新の際に、バッファをロックをしますが、書き込むサイズだけロックし、全データをロックしない様にしてください。

```
    g_WaveControllBGM->pDSBuffer->Lock( 0, play_size,
                                        //書き込むデータサイズ分だけロックする
                                        &pvAudioPtr1, &buff_size1,
                                        &pvAudioPtr2, &buff_size2,
                                        DSBLOCK_FROMWRITECURSOR
                                        //書き込み位置からバッファをロック
                                      );
```

ロックしたら、読み込んだデータをコピーしバッファを更新します。

この時、書き込むサイズに注意してください、更新するサイズがバッファの最後尾を超えて更新される場合、分割してデータコピーする必要があります。

書き込み位置が分割されているかどうかは、書き込みカーソルの2つ目がNULLかどうかで判断できます。

図11 －6－3 書き込み位置が分割されるケースのイメージ図

最後にロックを解除し、現在のカーソル位置を読み込んだカーソル位置として記録すれば終了です。

LIST 11 －6－1 BGMのストリーミング再生

```
//////////////////////////////////////////////////////////////////////
//ReadWaveFileOffset: WAVファイルの情報取得と読み込みを行う offset指定
//                    リニアPCMのみ対応、tagも単一のdataのみ対応
//
//char *pFileName:            ファイル名
//WAVEFORMATEX* pWaveFormat:  情報を格納する構造体へのポインタ
//char* pWaveData:            読み込まれるWAVデータへのポインタ
//int   ReadOffset:           読み込まれるWAVデータへのオフセット値
//int   ReadSize:             読み込まれるWAVデータのサイズ
//////////////////////////////////////////////////////////////////////
int ReadWaveFileOffset( char *pFileName, WAVEFORMATEX* pWaveFormat, char* 
pWaveData, int ReadOffset ,int ReadSize)
{
    FILE *fp;
    int wave_size;
    int data_top;               //データ先頭位置
    int read_size2;
    char dummy[16];

    fp = fopen( pFileName, "rb" );
    if ( !fp ) return( false );
```

```
//チャンク文字"RIFF"と、RIFFデータサイズを読み飛ばす
fread( dummy, 1, 4 + 4, fp );

//チャンク文字"WAVE"と、"fmt "を読み飛ばす
fread( dummy, 1, 4 + 4, fp );

//fmtデータサイズエリアを読み飛ばす
fread( dummy, sizeof( int ), 1, fp );

//フォーマット情報 取得リニアPCMデータのみで、拡張情報は存在しない
fread( pWaveFormat, sizeof( WAVEFORMATEX ) - 2, 1, fp );


//チャンク文字"data"を読み飛ばす
ZeroMemory(dummy, sizeof( dummy ));
while( strcmp( "data", dummy ) )
{
    fread( dummy, 1, 1, fp );
    if( dummy[0] == 'd' ) fread( &dummy[1], 1, 3, fp );
}

//Waveデータサイズ取得
fread( &wave_size, sizeof( int ), 1, fp );

//データ先頭位置を記録
data_top = ftell( fp );

//読み込み開始位置へシーク
//データ読み取り位置がデータ最終位置を越えないように調整
ReadOffset %= wave_size;
fseek( fp, ReadOffset, SEEK_CUR );

//waveデータ取得
  //読み取るデータがデータ最終位置を越えているか？

if( ReadOffset + ReadSize > wave_size )
{   //そうなら調整し、分割して読み込み
    read_size2 = wave_size - ReadOffset;
    fread( pWaveData, 1, read_size2, fp );
```

```
        //読み取り位置を、データの先頭へ戻し、残りデータを読み込む
        fseek( fp, data_top, SEEK_SET );
        fread( &pWaveData[read_size2], 1, ReadSize - read_size2, fp );
    }else
    {   //そうでなければ一括読み込み
        fread( pWaveData, 1, ReadSize, fp );
    }

    fclose( fp );

    return( wave_size );
}


/////////////////////////////////////////////////////
//11章－6　BGMのストリーミング再生
/////////////////////////////////////////////////////
void init11_06(TCB* thisTCB)
{
    int loop;
    int wave_size;
    //アクセス可能なバッファのサイズ
    DWORD  buff_size1, buff_size2;
    //WAVバッファアクセスポイントを格納する為のポインタ
    LPVOID pvAudioPtr1, pvAudioPtr2;

    //Waveファイルの情報とデータ取得
    wave_size = ReadWaveFileOffset(
                "..¥¥..¥¥data¥¥bgm01.wav",
                &g_WaveControllBGM->WaveFormat,
                g_WaveControllBGM->WaveData,
                0, BGM_READ_SIZE );

    //取得した情報を設定
    g_WaveControllBGM->DSBDesc.dwSize        = sizeof( DSBUFFERDESC );
    g_WaveControllBGM->DSBDesc.dwFlags       = DSBCAPS_LOCSOFTWARE;
    //バッファは余裕を見て読み込み量の3倍を設定
    g_WaveControllBGM->DSBDesc.dwBufferBytes = BGM_READ_SIZE * 3;
```

```
    g_WaveControllBGM->DSBDesc.lpwfxFormat    = &g_WaveControllBGM->WaveFormat;

    //サウンドバッファ作成
    g_pDSound->CreateSoundBuffer( &g_WaveControllBGM->DSBDesc,
                                  &g_WaveControllBGM->pDSBuffer,
                                  NULL );

    //最初のBGM再生
      //書き込むデータサイズ分だけロックする
    g_WaveControllBGM->pDSBuffer->Lock( 0, BGM_READ_SIZE,
                                        &pvAudioPtr1, &buff_size1,
                                        &pvAudioPtr2, &buff_size2,
                                        //書き込み位置からバッファをロック
                                        DSBLOCK_FROMWRITECURSOR
                                      );

    //サウンドデータをバッファへ書き込む
    memcpy( pvAudioPtr1, g_WaveControllBGM->WaveData, BGM_READ_SIZE );

    //ロック解除
    g_WaveControllBGM->pDSBuffer->Unlock( pvAudioPtr1, buff_size1,
                                          pvAudioPtr2, buff_size2 );

    //現在のカーソル位置を記録
    g_WaveControllBGM->pDSBuffer->GetCurrentPosition( NULL, &g_WaveControllBGM-
>BeforePlayCursor );
    g_WaveControllBGM->WaveFilePtr      = BGM_READ_SIZE;

}

void exec11_06(TCB* thisTCB)
{
    //アクセス可能なバッファのサイズ
    DWORD  buff_size1, buff_size2;
    //WAVバッファアクセスポイントを格納する為のポインタ
    LPVOID pvAudioPtr1, pvAudioPtr2;
    DWORD CurrentWriteCursor;
    RECT font_pos = {224, 192,640,480,};
    int play_size;
```

```
    //現在のカーソルを取得
    g_WaveControllBGM->pDSBuffer->GetCurrentPosition( NULL, &CurrentWriteCursor );

    //再生済みの容量を取得
    play_size = CurrentWriteCursor - g_WaveControllBGM->BeforePlayCursor;

    if( play_size < 0 )
    {
        play_size =
          (CurrentWriteCursor + BGM_READ_SIZE * 3) - g_WaveControllBGM-
>BeforePlayCursor;
    }
    //再生されたバッファが指定サイズ以上なら、バッファを更新
    if( play_size > BGM_READ_SIZE )
    {
        ReadWaveFileOffset(
        "..¥¥..¥¥data¥¥bgm01.wav",
        &g_WaveControllBGM->WaveFormat,
        g_WaveControllBGM->WaveData,
        g_WaveControllBGM->WaveFilePtr,
        play_size );

        g_WaveControllBGM->WaveFilePtr += play_size;


        //書き込むデータサイズ分だけロックする
        g_WaveControllBGM->pDSBuffer->Lock( 0, play_size,
                                    &pvAudioPtr1, &buff_size1,
                                    &pvAudioPtr2, &buff_size2,
                                   //書き込み位置からバッファをロック
                                    DSBLOCK_FROMWRITECURSOR
                                   );

        //WAVEデータの書き込み
        memcpy( pvAudioPtr1, g_WaveControllBGM->WaveData, buff_size1 );

        //書き込み位置が分割されているか？
        if( pvAudioPtr2 != NULL )
        {     //そうなら残りデータを書き込む
```

```
            memcpy( pvAudioPtr2,
                    &g_WaveControllBGM->WaveData[ buff_size1 ],
                    buff_size2 );
        }

        //ロック解除
        g_WaveControllBGM->pDSBuffer->Unlock( pvAudioPtr1, buff_size1,
                                              pvAudioPtr2, buff_size2 );

        //現在のカーソル位置を記録
        g_WaveControllBGM->BeforePlayCursor = CurrentWriteCursor;
    }

    if( g_DownInputBuff & KEY_Z )
        g_WaveControllBGM->pDSBuffer->Play( 0, 0, DSBPLAY_LOOPING );

    //停止
    if( g_DownInputBuff & KEY_X )
        g_WaveControllBGM->pDSBuffer->Stop();


    g_pFont->DrawText( NULL, "'Z'キーで曲再生¥n'X'キーで再生停止", -1, &font_pos,
DT_LEFT, 0xffffffff);

}
```

11-7 CD再生

用意されている関数を利用する

CDを再生する方法はいくつかありますが、簡単なのはマルチメディア関係を制御する関数、mciSendStringを使用することでしょう。

これは、CDの制御に対応する文字列を引数に渡す事によって、手軽にCDを制御できます。

```
mciSendString( 制御用のコマンド文字列, NULL, 0, NULL );
```

最初の引数にCD制御のコマンドとなる文字列を指定します。残りの引数は、コマンドの内容によって変化しますが、通常はNULLと0を渡せばOKです。

実際のプログラミング

初期化処理

さて、CDの再生ですが、初期化の処理として、まずCDデバイスをオープンするための文字列open cdaudioを送ります。

次に、CDの制御の単位がデフォルトでは秒単位で少し扱いにくいのでトラック単位に変更します。

```
//CDオープン(開始)
mciSendString( "open cdaudio", NULL, 0, NULL );

//トラック単位で制御
mciSendString("set cdaudio time format tmsf",NULL,0,NULL);
```

以上で初期化は終了です。

再生

次に再生ですが、いたって簡単で、コマンド文字列play cdaudioを送ります。

```
//再生
mciSendString( "play cdaudio", NULL, 0, NULL );
```

これで、CDが再生されます。

その他の制御用コマンド文字列

このコマンドには他にも様々な種類がありますが、代表的なものをいくつか記述します。

図11－7－1 代表的なコマンド

代表的なコマンド

play cdaudio	再生
stop cdaudio	停止
play cdaudio notify	CDの先頭から再生
set cdaudio door open	CDトレイを開く
set cdaudio door closed	CDトレイを閉じる
status cdaudio current track	現在のトラック番号を取得する
open cdaudio	CD制御を開始する
close cdaudio	CD制御を停止する

以上のコマンドで、status cdaudio current trackコマンドは引数を要求するので少し詳しく解説します。

このコマンドは現在再生中のトラック番号を調べる命令で、コマンド以外に2つの引数を要求します。

2番目の引数には、再生中のトラック番号を格納するためのバッファへのポインタを指定し、3番目の引数は格納するバッファのサイズを指定します。

```
//現在のトラックを取得
    mciSendString("status cdaudio current track", current_track ,
sizeof(current_track) , NULL);
```

これで現在再生中のトラック番号が、current_trackに文字列として格納されます。

なお、これらのCD制御が終わり、プログラムを終了させる際にはclose cdaudioコマンドを送り、クローズする事を忘れないようにしてください。

LIST 11 −7−1 CDDA再生

```
void init11_07(TCB* thisTCB)
{
    //CDオープン(開始)
    mciSendString( "open cdaudio", NULL, 0, NULL );

    //トラック単位で制御
    mciSendString("set cdaudio time format tmsf",NULL,0,NULL);

}

void exec11_07(TCB* thisTCB)
{
    RECT font_pos = {160, 192,640,480,};
    char current_track[16];

    g_pFont->DrawText( NULL, "'Z'キー　曲再生¥n'X'キー　再生停止", -1, &font_pos,
DT_LEFT, 0xffffffff);
    font_pos.top += 48;
    g_pFont->DrawText( NULL, "カーソルキー", -1, &font_pos, DT_LEFT,
0xffffffff);
    font_pos.top += 16;
    g_pFont->DrawText( NULL, "左　CDトレイオープン", -1, &font_pos, DT_LEFT,
0xffffffff);
    font_pos.top += 16;
    g_pFont->DrawText( NULL, "右　CDトレイクローズ", -1, &font_pos, DT_LEFT,
0xffffffff);

    //現在のトラックを取得
    mciSendString("status cdaudio current track", current_track ,
sizeof(current_track) , NULL);

    font_pos.top += 16;
    g_pFont->DrawText( NULL, "現在のトラック", -1, &font_pos, DT_LEFT, 0xffffffff);
    font_pos.left += 160;
    g_pFont->DrawText( NULL, current_track, -1, &font_pos, DT_LEFT, 0xffffffff);


    //先頭から再生
    if( g_DownInputBuff & KEY_UP     )
```

```
        mciSendString( "play cdaudio notify", NULL, 0, NULL );

    //CDトレイオープン
    if( g_DownInputBuff & KEY_LEFT )
        mciSendString( "set cdaudio door open", NULL, 0, NULL );

    //CDトレイクローズ
    if( g_DownInputBuff & KEY_RIGHT)
        mciSendString( "set cdaudio door closed", NULL, 0, NULL );

    //再生
    if( g_DownInputBuff & KEY_Z     )
        mciSendString( "play cdaudio", NULL, 0, NULL );

    //停止
    if( g_DownInputBuff & KEY_X     )
        mciSendString( "stop cdaudio", NULL, 0, NULL );
}
```

Chapter 12

その他

12-1 カードゲームのデータ管理 1

LIST 構造で管理する

麻雀を始めとした各種テーブルゲームやトランプ等のカードゲームでは多数のカードを管理する必要があります。

この際のデータ管理として、配列を用いてもいいのですが、管理のしやすさからLIST構造を用いるのが一般的です。

簡単なカード管理プログラム

ここではトランプを管理する、簡単なカード管理プログラムを作成してみましょう。

管理プログラムは5つの関数と、カードデータから構成されています。

関数は、「データの初期化」「リストへのカードの追加」「リストへのカードの削除」「Index からのカードの取得」「トランプの種類によるカードの検索」の5種類。

データは、カードの種類、表示データを保持します。

初期化

まずは、初期化から見ていきます。行なう初期化は、カードデータの作成と、トランプの画像表示用データの作成です。

カードデータはリスト構造になっており、以下の様になっています。

```
typedef struct CARD_STRUCT{
    CARD_STRUCT*    Prev;           //前のカード
    CARD_STRUCT*    Next;           //次のカード
    int             CardType;       //カードタイプ
    int             CardNo;         //カードナンバー
    RECT            CardChipData;   //カードの表示データ
} EX12_CARD_STRUCT;
```

Prev と Next はそれぞれリスト接続された前のカードと次のカードを示し、NULL で先頭または終端になります。

CardType は、スペードやクラブ等、カードの種類を格納し、CardNo はカードの数字を格納します。

CardChipDataは、表示の元画像を切り分けて表示するためのデータで、RECT構造体です。

図12－1－1 トランプの画像データ

♠A	♠2	♠3	♠4	♠5	♠6	♠7	♠8	♠9	♠10	♠J	♠Q	♠K
♣A	♣2	♣3	♣4	♣5	♣6	♣7	♣8	♣9	♣10	♣J	♣Q	♣K
♥A	♥2	♥3	♥4	♥5	♥6	♥7	♥8	♥9	♥10	♥J	♥Q	♥K
♦A	♦2	♦3	♦4	♦5	♦6	♦7	♦8	♦9	♦10	♦J	♦Q	♦K

データはトランプ画像データのに対応する形で作成されます。

画像の配置はあらかじめシンプルに並べてありますので、カードデータはループの数値を、表示データはカードの幅と高さをそれぞれ乗算してやることで作成出来ます。

LIST 12－1－1

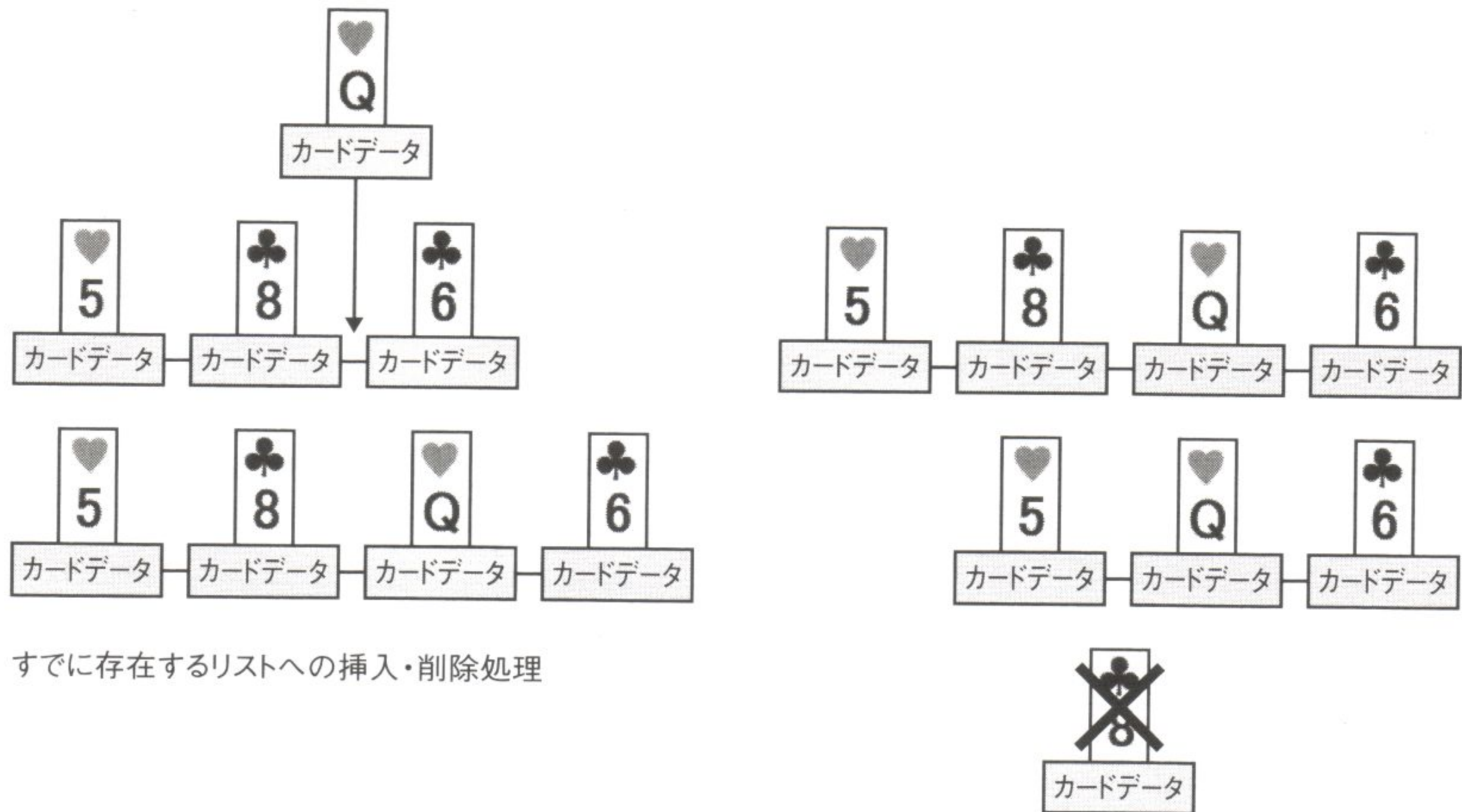
```
typedef struct CARD_STRUCT{
    CARD_STRUCT*    Prev;
    CARD_STRUCT*    Next;
    int             CardType;
    int             CardNo;
    RECT            CardChipData;
} EX12_CARD_STRUCT;

static EX12_CARD_STRUCT CardData[CARD_TOTAL];

/////////////////////////////////////////////////////
//カードの初期化
/////////////////////////////////////////////////////
void CardInit( void )
{
    int loopX;
    int loopY;
    //カードデータの初期化
    for (loopY = 0; loopY < CARD_TYPE; loopY++)
    {
        for (loopX = 0; loopX < CARD_COUNT; loopX++)
```

```
        {
            //カードデータの初期化
            CardData[ loopY * CARD_COUNT + loopX ].Prev     = NULL;
            CardData[ loopY * CARD_COUNT + loopX ].Next     = NULL;
            CardData[ loopY * CARD_COUNT + loopX ].CardType = loopY;
            CardData[ loopY * CARD_COUNT + loopX ].CardNo   = loopX;

            //カード表示データの初期化
            CardData[ loopY * CARD_COUNT + loopX ].CardChipData.top
                = CARD_HEIGHT * loopY;

            CardData[ loopY * CARD_COUNT + loopX ].CardChipData.bottom
                = CARD_HEIGHT * loopY + CARD_HEIGHT;

            CardData[ loopY * CARD_COUNT + loopX ].CardChipData.left
                = CARD_WIDTH  * loopX;

            CardData[ loopY * CARD_COUNT + loopX ].CardChipData.right
                = CARD_WIDTH  * loopX + CARD_WIDTH;
        }
    }
}
```

カードを管理する関数

次は実際のカードの管理関数です。

管理のための関数は２種類で、カードのリストへの登録を行なうCardAddとリストからの削除を行なうCardDelです。

図12-1-2 カードのリストへの挿入と削除のイメージ図

すでに存在するリストへの挿入・削除処理

カードを追加する関数

CardAdd関数は、リストの指定された位置にカードを挿入します。

この際、すでにリストが作成済みであることが必要です。

そのため、最初のカードリストは手動で作成する必要がある事に注意してください。

内部処理自体は単純で、カードの挿入位置を検索後、検索位置に指定のカードを挿入します。

ただし、リストの最終位置への挿入はNULLポインタを扱うため、特別扱いをしています。

LIST 12-1-2 カードの追加

```
void CardAdd( EX12_CARD_STRUCT* pCardList, EX12_CARD_STRUCT* pCard, int
Index)
{
    int loop;
    EX12_CARD_STRUCT* pCardListOld = pCardList;

    for( loop = 0; loop < Index; loop++ )
    {//カード挿入位置の検索
        //1つ前のカードを記録しておく
        pCardListOld = pCardList;

        pCardList = pCardList->Next;
    }
```

```
    if( pCardList != NULL )
    {//リストへの挿入
        pCard->Prev = pCardList;
        pCard->Next = pCardList->Next;
        pCardList->Next->Prev = pCard;
        pCardList->Next = pCard;
    } else {
     //リスト終端位置への挿入
        //1つ前のカードの後尾に挿入する
        pCardList = pCardListOld;
        pCard->Prev = pCardList;
        pCard->Next = NULL;
        pCardList->Next = pCard;
    }

}
```

カードを削除する関数

次にCardDel関数です。この関数は、リストの指定された位置のカードを削除します。

この関数も、CardAdd関数と同様に、削除位置のカードを検索後、カードを登録から削除します。

リスト終端を特別扱いする所も同様ですが、削除関数の場合、リストの先頭を削除する事があるのでそこも特別扱いにしています。

また先頭を削除した際、リストの開始位置が変わってしまうので、返り値としてリストの先頭へのポインタを返しています。

LIST 12－1－2 カードの削除

```
EX12_CARD_STRUCT* CardDel( EX12_CARD_STRUCT* pCardList, int Index)
{
    int loop;
    EX12_CARD_STRUCT* pCardListOld = pCardList;
    EX12_CARD_STRUCT* pCardNext;

    for( loop = 0; loop < Index; loop++)
    {//削除カードの検索
        pCardList = pCardList->Next;
    }
```

```
    if( Index == 0 )
    {//リスト先頭からの削除
        pCardNext = pCardList->Next;
        pCardNext->Prev = NULL;
        pCardList->Prev = NULL;
        pCardList->Next = NULL;

        //削除した次のカードがリストの先頭
        return pCardNext;
    }

    if( pCardList->Next != NULL )
    {//リストからの削除
        pCardList->Prev->Next = pCardList->Next;
        pCardList->Next->Prev = pCardList->Prev;
        pCardList->Prev = NULL;
        pCardList->Next = NULL;
    } else {
     //リスト終端位置からの削除
        pCardList->Prev->Next = NULL;
        pCardList->Prev = NULL;
        pCardList->Next = NULL;
    }

    //リストの先頭アドレス
    return pCardListOld ;
}
```

12-2 カードゲームのデータ管理2

カードを取得する関数

カードの初期化、管理のプログラムに続いて、カードの取得、検索関数を作成します。

カードを取得する関数CardGetは指定位置のカードへのポインタを返します。

プログラムは単純なループとなっており、指定回数だけリストを辿って、カードへのポインタを返します。

LIST 12－2－1 カードの取得

```
EX12_CARD_STRUCT* CardGet( EX12_CARD_STRUCT* pCardList, int Index)
{
    int loop;
for( loop = 0; loop < Index; loop++)
    {//指定番号のカードの検索
        pCardList = pCardList->Next;
    }
return pCardList;
}
```

カードを検索する関数

検索関数CardSearchは指定したカード位置のIndexを返します。

プログラムは渡されたリストを、指定されたカードが発見されるまで辿っていきます。

カードが発見された場合はIndex値を返し、カードがなければ－1を返します。

両関数ともリストを辿るループが基本となっており、そう難しいところは無いでしょう。

LIST 12－2－2 カードの検索

```
int CardSearch( EX12_CARD_STRUCT* pCardList, int CardType, int CardNo)
{
    int count = 0;
    while(pCardList != NULL)
    {//条件に合わせてカードを検索
        if( (pCardList->CardType == CardType) &&
            (pCardList->CardNo == CardNo) )
```

```
        {//カードを発見したらIndexを返す
            return count;
        }
        count++;
        pCardList = pCardList->Next;
    };
    //カードがなければー1を返す
    return -1;
}
```

管理関数の使い方

最後に、管理関数の基本的な使い方として、全てのカードを表示するプログラムをみてみます。

初期化

初期化部分の始めは、カードの画像データ読み込みと、カードの初期化関数を呼び出しています。その後、メインとなるカードのリストを作成しています。

カードリストのポインタは、カードの表示中はずっと保持されるため、タスク上に作成しています。

作成自体は単純で、カードの最初の1枚をポインタへ格納するだけです。

作成後は、残りのカードをAddCardを用いてリストへ登録しています。

LIST 12 ー2ー3

```
typedef struct{
    EX12_CARD_STRUCT*     MainCard;
} EX12_01_STRUCT;
void init12_01(TCB* thisTCB)
{
    EX12_01_STRUCT* work = (EX12_01_STRUCT*)thisTCB->Work;
    int loop;
//使用するテクスチャの読み込み
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥card.png",&g_pTex[0] );
//カードの初期化
    CardInit();
//最初のカードを登録
    work->MainCard = &CardData[0];
    //残りのカードをカードリストに登録
    for( loop = 1; loop < CARD_TOTAL; loop++)
```

```
    {
        CardAdd( work->MainCard , &CardData[ loop ],loop );
    }
}
```

表示処理

次に実際の表示処理です。

処理自体はリストの通りに表示するだけですので、リストを辿るループになります。

表示座標を計算したあと、カードデータの画像表示情報を元にスプライトを作成、表示しています。リストの最後まで表示したら終了になります。

LIST 12－2－4

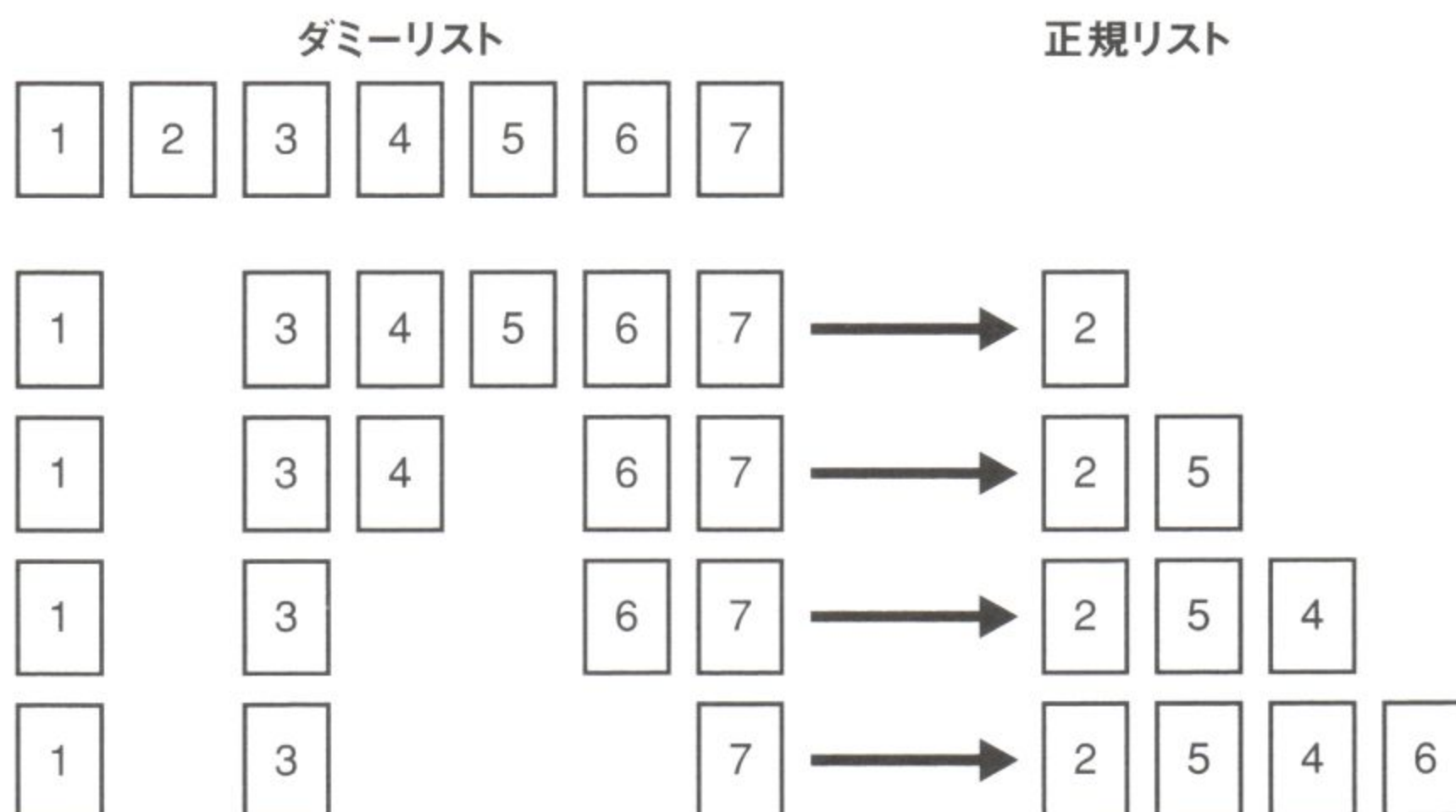
```
void exec12_01(TCB* thisTCB)
{
    EX12_01_STRUCT* work = (EX12_01_STRUCT*)thisTCB->Work;
    int index = 0;
    EX12_CARD_STRUCT* pchk_card;
    SPRITE2  card;

    pchk_card = work->MainCard;
    while( pchk_card != NULL )
    {
        //カードの表示座標を計算
        card.X = (index % CARD_COUNT) * CARD_WIDTH;
        card.Y = (index / CARD_COUNT) * CARD_HEIGHT;
        //カードの表示
        card.SrcRect = &pchk_card->CardChipData;
        SpriteDraw2( &card, 0);
        index++;
        //次のカード
        pchk_card = pchk_card->Next;
    };
}
```

12-3 カードをシャッフルする

シャッフルする処理の概要

麻雀ゲームやカードゲームにおいて、牌を混ぜたりカードを切る処理は必須です。
ここではその処理を作成してみましょう。
基本となる処理は、[12－1]で作成したカード管理プログラムを使用します。

処理の概要ですが、まずカードをそのままの順番で登録したダミーのリストを作成します。
作成後、そのリストからカードをランダムで1枚抜き出し、正規のリストへ登録します。
これをカードの枚数分だけ繰り返せば、正規のリストへシャッフルされたデータが登録されます。

図12－3－1 ダミーのリストを作成し、正規のリストへ再登録するイメージ図

一度順番に並んだダミーのリストを作り、
そこからランダムに削除したカードを登録していく

カードをシャッフルするプログラム

ダミーリストの作成

では実際の処理を見ていきます。
カード管理プログラムの初期化後、ダミーとなるリストを作成します。
ダミーリストは単純にカードを並べるだけなので、ループ回数をカードの登録キーにして作成します。

正規リストの作成

ダミーリストの作成後、正規のリストを作成します。

まずダミーのリストから抜き出す最初の１枚をランダムで選びます。

次に選んだカードのポインタを関数CardGetで取得し、保持しておきます。

その後、抜き出したカードをダミーから削除し、保持していたカードを正規リストの１枚目として登録して、正規リストの作成は終了です。

残りのカードですが、こちらも同様の手段で正規リストに登録していきます。

注意点として、カードを削除したダミーリストの枚数は当然元のカード枚数より減っていますので、乱数の幅をカードの残り枚数に合わせて狭めていく必要があります。

また、ダミーリストに残った最後のカードは削除できないため、特別扱いで登録しています、この点も注意してください。以上で、シャッフルの処理は終了です。

最後に本当にシャッフルがされているかを表示して確認します。

こちらの処理は[12－2]の表示処理とほぼ同一ですので、詳しくはそちらを参考にしてください。

LIST 12－3－1

```
typedef struct{
    EX12_CARD_STRUCT*     MainCard;
} EX12_03_STRUCT;

void init12_03(TCB* thisTCB)
{
    EX12_03_STRUCT* work = (EX12_03_STRUCT*)thisTCB->Work;
    EX12_CARD_STRUCT*     tmpCardList;
    EX12_CARD_STRUCT*     padd_card;
    int card_id;
    int loop;
    //使用するテクスチャの読み込み
    D3DXCreateTextureFromFile( g_pD3Ddevice, "..¥¥..¥¥data¥¥card.png",&g_pTex[0] );
    //カードの初期化
    CardInit();
    //シャッフル用にカードをリストに登録
    tmpCardList = &CardData[0];
    for( loop = 1; loop < CARD_TOTAL; loop++)
    {
        CardAdd( tmpCardList , &CardData[ loop ],loop );
    }
    //最初の1枚　リストから乱数でカードを1枚抜き出す
```

```
    card_id = rand() % CARD_TOTAL;
    padd_card = CardGet( tmpCardList,card_id );
    tmpCardList = CardDel( tmpCardList     , card_id );
    //抜き出したカードを正式なリストに登録
    work->MainCard = padd_card;

    //残りのカードも同様に、乱数で抜き出して正式なリストに登録
    for( loop = 1; loop < CARD_TOTAL - 1; loop++ )
    {
        card_id = rand() % (CARD_TOTAL - loop - 1);          //指定範囲の乱数を得る
        padd_card = CardGet( tmpCardList,card_id );
        tmpCardList = CardDel( tmpCardList     , card_id );
        CardAdd( work->MainCard , padd_card,loop );
    }
    //最後のカードを登録
    CardAdd( work->MainCard , tmpCardList, 1 );
}

void exec12_03(TCB* thisTCB)
{
    EX12_03_STRUCT* work = (EX12_03_STRUCT*)thisTCB->Work;
    int index = 0;
    EX12_CARD_STRUCT* pchk_card;
    SPRITE2  card;

    pchk_card = work->MainCard;
    //リストを辿ってカードを表示
    while( pchk_card != NULL )
    {
        //カードの表示座標を計算
        card.X = (index % CARD_COUNT) * CARD_WIDTH;
        card.Y = (index / CARD_COUNT) * CARD_HEIGHT;
        //カードの表示
        card.SrcRect = &pchk_card->CardChipData;
        SpriteDraw2( &card, 0);
        index++;
        //次のカード
        pchk_card = pchk_card->Next;
    }
}
```

12-4 役を判定する

役を判定する処理手順

役を判定する処理はテーブルゲームなどにおいて、ゲームの中心となる処理の１つです。

けれども、こういった役を判定する処理はゲームによって千差万別で、それゆえに汎用的な処理を作る事はかなり難しくなります。

また仮に作ったとしても、扱いそのものが難しくなる事も多く、結局ゲームに合わせてそのつど作成する事が一番という形になってしまいます。

ポーカーの役判定ルーチン

とはいえ、一応形に添った手順は存在します。ここではその手順を踏まえ、トランプのポーカーを例に役の判定ルーチンを作成してみます。

図 12－4－1 役の情報収集と、判定を行なう部分

諸条件チェック部		役判定部
同じ数のカードは何枚あるか?		1:同じマークのカードが5枚ある→フラッシュ
同じマークのカードは何枚あるか?	→	2:数字が連続して5 枚並んでいる→ストレート
カードは連続してならんでいるか?		3:1と2が、同時に成立している→ストレートフラッシュ

大きく2 つに分ける
まず、諸条件を元にチェックしフラグを立てる部分

その後、フラグを元に役を判定する部分

図 12－4－2 ポーカーの役の図

ワンペア	同じ数字が2枚
ツーペア	ワンペアが2組
スリーカード	同じ数字が3枚
ストレート	数字が連続して5枚並ぶ

フルハウス	ワンペア1 組とスリーカード1組
フラッシュ	同じマークが5枚
ストレートフラッシュ	ストレートで、かつフラッシュ
フォーカード	同じ数字が4 枚

基本となる考え方は、役を構築する基本情報の収集と、集めた情報を元に役を判定する部分に分ける事です。

ポーカーで言えば、ペアの数や、同じカードが何枚あるか？などが、基本情報となります。

これらを収集し、判定部分で扱い易い形に加工してやります。

役判定プログラム

では実際にプログラムを見ていきます。

収集する情報の決定

まず、基本情報の種類を決定します。ポーカーの役は幾つかの役の組み合わせで構成されている事が多く、ここでは４種類の情報を収集、作成します。

・情報の内訳ですが、まず、幾つペアがあるかをカウントする、pair_count
・同じ数のカードが手札に最大何枚含まれているかを示す、max_count
・カードが連番になっている場合、それがを幾つならんでいるかを数える straight
・同じ種類のカードが何枚あるかを数える flash

情報収集作業エリアの準備

次に、情報を収集するための作業エリアとして、card_NoCount と card_TypeCount を用意し、手札のカードの種類と数をカウントします。

この処理はリストを辿っていき、カードの種類に合わせて配列をカウントしてやるだけです。

役の情報収集

次は役の情報を収集します。収集する数が比較的多いので、ここでは個別に解説せず箇条書きにします。

pair_count　同数のカードの枚数が２の時にカウントされます。
max_count　同数のカードの最大所持枚数を記録します。
straight　現在チェックしているカードとループの１つ前のカードをチェックして両方手札にあればカウントされる。
ただし、０番目のカードだけは１つ前のカードが存在しないのでカウントしません。
flush　同種のカードの最大所持枚数を記録します。

役の判定

最後に収集した情報を元に、役の判定を行ないます。

ここでも、数が多いので判定の解説は箇条書きにします。

判定自体はそれほど難しい物はありません。

情報収集の条件、及びリストとを見比べて、収集した情報を元に判定するという部分を確実に理解してください。

ワンペア	pair_count が 1 の時はワンペア
ツーペア	pair_count が 2 の時はツーペア
スリーカード	max_count が 3 のときはスリーカード
ストレート	straight が 4 の時はストレート(4 回連続しているので 5 枚)
フラッシュ	flush が 5 の時はフラッシュ(手札は 5 枚なので 5 の時は全てのカードが同種)
フルハウス	pair_count が 1 で max_count が 3 のときはフルハウス(ワンペアとスリーカードの条件が重なった時)
フォーカード	max_count が 4 のときはフォーカード
ストレートフラッシュ	straight が 4 で flush が 5 の時はストレートフラッシュ(ストレートとフラッシュの条件が重なった時)

LIST 12 ―4―1 役を判定するには

```
typedef struct{
    EX12_CARD_STRUCT*     MainCard;
} EX12_04_STRUCT;
//手札の所持枚数
#define HAND_CARD_COUNT 5

void EX12_04_CardShuffle( EX12_04_STRUCT* CardWork )
{
    EX12_CARD_STRUCT*     padd_card;
    EX12_CARD_STRUCT*     tmpCardList;
    int card_id;
    int loop;

    //カードデータの初期化
    for (loop = 0; loop < CARD_TOTAL; loop++ )
    {   //カードデータの初期化
        CardData[ loop ].Prev      = NULL;
        CardData[ loop ].Next      = NULL;
    }

    //シャッフル用にカードをリストに登録
    tmpCardList = &CardData[0];
```

```
    for( loop = 1; loop < CARD_TOTAL; loop++)
    {
        CardAdd( tmpCardList , &CardData[ loop ],loop );
    }

    //最初の1枚　リストから乱数でカードを1枚抜き出す
    card_id = rand() % CARD_TOTAL;
    padd_card = CardGet( tmpCardList,card_id );
    tmpCardList = CardDel( tmpCardList    , card_id );
    //抜き出したカードを手札のリストに登録
    CardWork->MainCard = padd_card;

    //残りのカードも同様に、乱数で抜き出して手札のリストに登録
    for( loop = 1; loop < HAND_CARD_COUNT; loop++ )
    {
        card_id = rand() % (CARD_TOTAL - loop - 1);
        padd_card = CardGet( tmpCardList,card_id );
        tmpCardList = CardDel( tmpCardList    , card_id );
        CardAdd( CardWork->MainCard , padd_card,loop );
    }
}

void init12_04(TCB* thisTCB)
{
    EX12_04_STRUCT* work = (EX12_04_STRUCT*)thisTCB->Work;

    //使用するテクスチャの読み込み
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥card.png",&g_pTex[0] );

    //カードの初期化
    CardInit();

    EX12_04_CardShuffle( work );

}
void exec12_04(TCB* thisTCB)
{
    EX12_04_STRUCT* work = (EX12_04_STRUCT*)thisTCB->Work;
    int loop;
```

```
EX12_CARD_STRUCT* pchk_card;
SPRITE2  card;
RECT font_pos = {160, 192,640,480,};

char card_NoCount[ CARD_COUNT ];
char card_TypeCount[ CARD_TYPE ];
char pair_count = 0;
char max_count = 0;
char straight = 0;
char flush = 0;


//キー入力で再度シャッフルして手札を配る
if( g_DownInputBuff & KEY_Z     )
{
    EX12_04_CardShuffle( work );
}

//手札の表示
pchk_card = work->MainCard;
for( loop = 0; loop < HAND_CARD_COUNT ; loop++ )
{
    //カードの表示座標を計算
    card.X = loop * CARD_WIDTH + 128;
    card.Y = 128;

    card.SrcRect = &pchk_card->CardChipData;
    SpriteDraw( &card, 0);
    pchk_card = pchk_card->Next;
}

//チェック用のカウンタを初期化する
for( loop = 0;loop < CARD_TYPE ;loop++ )
    card_TypeCount[ loop ] = 0;
for( loop = 0;loop < CARD_COUNT;loop++ )
    card_NoCount[ loop ] = 0;

//手札の種類をカウントする
pchk_card = work->MainCard;
for( loop = 0;loop < HAND_CARD_COUNT ;loop++ )
```

```
{
    card_NoCount[ pchk_card->CardNo ]++;
    card_TypeCount[ pchk_card->CardType ]++;
    pchk_card = pchk_card->Next;
}

//役の情報を集める
for( loop = 0;loop < CARD_COUNT ;loop++ )
{
     //ペアの数をカウントする
     if( card_NoCount[ loop ] == 2 )pair_count++;

     //最大数のカードNoをカウントする
     if( card_NoCount[ loop ] > max_count )
         max_count = card_NoCount[ loop ];

     //カードが何回連続しているかをカウントする
     if( loop != 0)      //最初の1枚目はカウント出来ない
     {
         if( card_NoCount[ loop - 1 ] != 0 &&
             card_NoCount[ loop ] != 0       )
         {
             straight++;
         }
     }
}
for( loop = 0;loop < CARD_TYPE ;loop++ )
{ //同種のカードが最大何枚あるかをカウントする
     if( card_TypeCount[ loop ] > flush )
         flush = card_TypeCount[ loop ];
}

//集めた情報を元に実際の役を判定

//ワンペア
if( pair_count == 1 )
{
    g_pFont->DrawText( NULL,
        "ワンペア", -1, &font_pos, DT_LEFT, 0xffffffff);
    font_pos.top += 24;
```

```
    }

    //ツーペア
    if( pair_count == 2 )
    {
        g_pFont->DrawText( NULL,
            "ツーペア", -1, &font_pos, DT_LEFT, 0xffffffff);
        font_pos.top += 24;
    }

    //スリーカード
    if( max_count == 3 )
    {
        g_pFont->DrawText( NULL,
            "スリーカード", -1, &font_pos, DT_LEFT, 0xffffffff);
        font_pos.top += 24;
    }

    //ストレート
    if( straight == 4 )
    {
        g_pFont->DrawText( NULL,
            "ストレート", -1, &font_pos, DT_LEFT, 0xffffffff);
        font_pos.top += 24;
    }

    //フラッシュ
    if( flush == 5 )
    {
        g_pFont->DrawText( NULL,
            "フラッシュ", -1, &font_pos, DT_LEFT, 0xffffffff);
        font_pos.top += 24;
    }

    //フルハウス
    if( pair_count == 1 && max_count == 3 )
    {
        g_pFont->DrawText( NULL,
            "フルハウス", -1, &font_pos, DT_LEFT, 0xffffffff);
        font_pos.top += 24;
```

```
    }

    //フォーカード
    if( max_count == 4 )
    {
        g_pFont->DrawText( NULL,
            "フォーカード", -1, &font_pos, DT_LEFT, 0xffffffff);
        font_pos.top += 24;
    }

    //ストレートフラッシュ
    if( straight ==  4 && flush == 5 )
    {
        g_pFont->DrawText( NULL,
            "ストレートフラッシュ", -1, &font_pos, DT_LEFT, 0xffffffff);
        font_pos.top += 24;
    }
}
```

12-5 ランダム要素の使い方

ゲームにおけるランダム要素

ゲームにおいてランダムな要素は微妙な立場にあります。

例えば、敵の行動や、弾の発射方向など、特定のパターンを持った行動に対してランダム要素を適用すると、あたかも複数のパターンがあるかのように見せる事が出来ます。

ところが、この適用が行き過ぎると、相手の動きがまったく予想できなかったり、突然意味もなく死んでしまう…などといった事になってしまいます。

乱数は、手軽にデータを作成したり、難易度の調整をするのにも使える便利な手法ですが、あまりあちこちで使用するのは考えものです。

ゲームは、プレイヤーが攻略のために、あれこれ試行錯誤をし、判断する事にその面白さが集約されます。

ユーザーがゲームを攻略のためにベストな行動や判断したのにもかかわらず、その結果がランダムで決まってしまうのでは、やる気を著しく損なってしまいます。

ランダムを使うのであれば、あくまでユーザーが納得できる範囲内で使用するべきです。

もちろん、本質的には、ランダム要素に頼らずに作る事が望ましいのですが、ゲームによってはランダム要素を使用しない事は、かなりの困難を伴いますので中々難しい所です。

関数randを拡張する

さて、このランダム要素を作成するのに不可欠な関数randですが、この関数で使える乱数は整数のみとなっています。

しかも、好きな範囲が指定できるわけではなく、0～RAND_MAXの間の数値を返す事になっています。

このままでは少々使い勝手が悪いので、マクロを用いて、いくつか拡張をして見ましょう。

まずは好きな範囲の数を扱えるようにしてみます。

```
//好きな範囲の乱数
#define rand_range( a, b ) ( (rand() % ( b - a + 1 )) + a )
```

引数aとbの間の乱数を求めます。

割余を用いて、乱数を指定の範囲内に丸めていますが、そのままだと１つ少ない値になってしまうので、丸める時に＋１をしています。

次に小数点化です。

整数型を float 型 へキャストしています。特に問題になるところは無いでしょう。

```
//小数点化 0～1.0の間の数値を返す
#define rand_float() ((float)rand() / (float)RAND_MAX)
```

最後に、好きな範囲の乱数を小数点で返すようにしてみます。

上記の小数点化のマクロを用いていますので注意してください。その点以外は整数版とほぼ同じです。

```
//好きな範囲の乱数　小数点版
#define rand_range_float( a, b ) ( ( rand_float() * ( b - a )) + a )
```

LIST 12 －5－1 ランダム

```
void exec12_05(TCB* thisTCB)
{
//タスクワーク上に値を保存する
    int*   int_work = thisTCB->Work;
    float* float_work1 = (float*)&thisTCB->Work[1];
    float* float_work2 = (float*)&thisTCB->Work[2];

    RECT font_pos = {160, 192,640,480,};
    char str[256];

    //キー入力で乱数値を再計算
    if( g_DownInputBuff & KEY_Z    )
    {
         *int_work    = rand_range( 25, 50 );
         *float_work1 = rand_float();
         *float_work2 = rand_range_float( 3, 17 );
    }

    g_pFont->DrawText( NULL,
        "'Z'キー　で乱数値を計算", -1, &font_pos, DT_LEFT, 0xffffffff);
    font_pos.top += 48;
```

```
    g_pFont->DrawText( NULL,
        "範囲25－50の間の乱数", -1, &font_pos, DT_LEFT, 0xffffffff);
    font_pos.top += 16;
    sprintf( str,"%d", *int_work );
    g_pFont->DrawText( NULL, str, -1, &font_pos, DT_LEFT, 0xffffffff);
    font_pos.top += 32;

    g_pFont->DrawText( NULL,
        "小数点の乱数", -1, &font_pos, DT_LEFT, 0xffffffff);
    font_pos.top += 16;
    sprintf( str,"%f",*float_work1 );
    g_pFont->DrawText( NULL, str, -1, &font_pos, DT_LEFT, 0xffffffff);
    font_pos.top += 32;


    g_pFont->DrawText( NULL,
        "範囲3－27の間の乱数 小数点", -1, &font_pos, DT_LEFT, 0xffffffff);
    font_pos.top += 16;
    sprintf( str,"%f",*float_work2 );
    g_pFont->DrawText( NULL, str, -1, &font_pos, DT_LEFT, 0xffffffff);
    font_pos.top += 32;
}
```

12-6 敵車の動き

コース上を走る

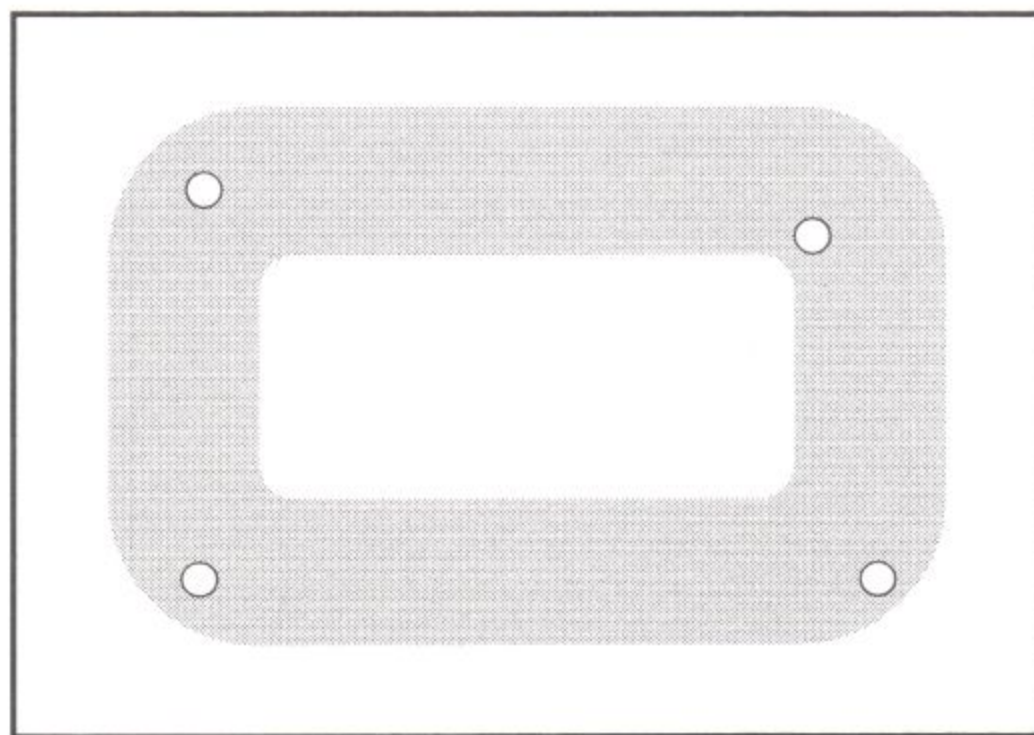
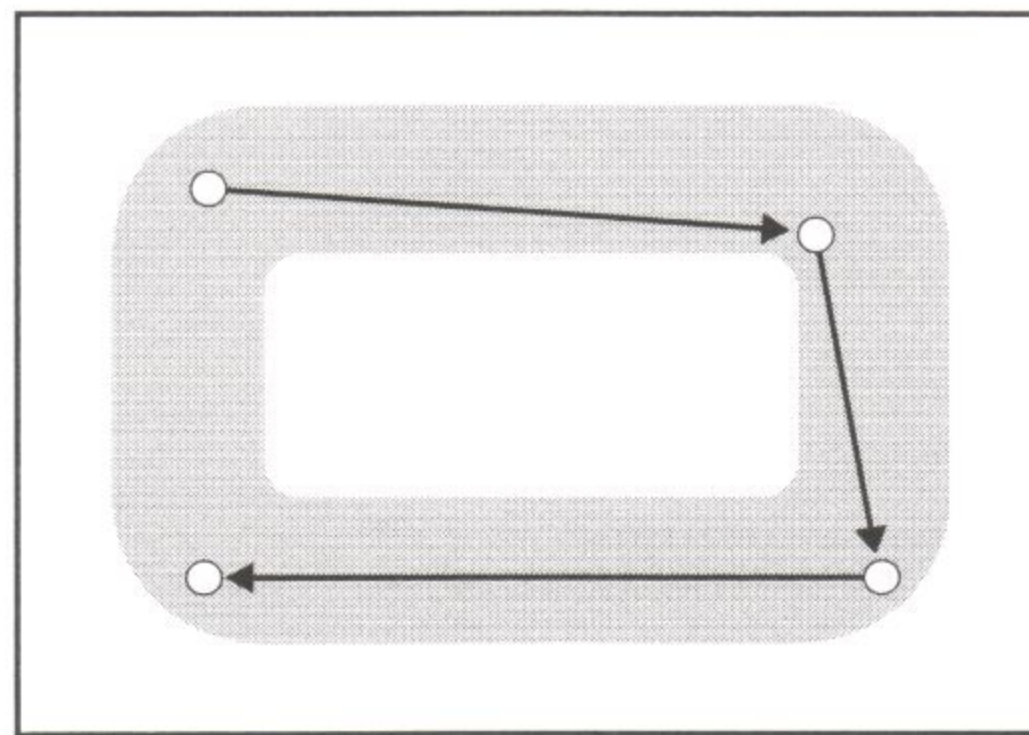
レースゲーム等での敵車の動きを考えてみます。

こういう動きを考える時に、問題になるのは、コース上ををどうやって進んだら良いかという事でしょう。

一番手軽な方法は、コース上に車の目標となる、チェックポイントのデータを配置し、それに向かって進むようにする事です。

この手法であれば、どのような複雑なコースでもポイントを細かく配置すれば、コースどおりに進めますし、いわゆるコース取りや、ライン取りといった、コースの状況にあわせた移動も可能になります。

また他にも、チェックポイントに向かって移動する方法やアルゴリズムによって、様々な動きが可能になります。

図12－6－1 コース上にポイントを配置

コース上にチェックポイントを配置して

それぞれのポイントを目指すように動かす

チェックポイントによる管理

実際のプログラムをみていきます。処理の内容は、コース上を車に見立てたボールが移動するものです。

はじめにチェックポイントを通過しているかチェックします。

サンプルでは、時間により管理しているのでタイマーをチェックしていますが、実際には当たり判定を元にチェックする事が多くなると思います。

次にチェックポイントを通過したら、次のチェックポイントへ向かう処理を行ないます。

まず、次のポイントへの座標を取得し、設定を行ないます。

そして、ポイントからの相対値を求め、次にatan2関数を用いてポイントへの方向を取得します。

その後、時間と距離から速度を割り出し、目標への増分値を取得します。

なお、サンプルでの動きは、等速運動になるため、動きがかなり直線的です。この辺は6章を参考に自由に置き換えてください。

最後に移動の為、ボールへ増分値を加算してやれば、処理は終了です。

LIST 12－6－1 敵車の動き

```
//目標となるポイントの数
#define POINT_MAX 4
#define START_X   320
#define START_Y   96
#define MOVE_TIME 60.0

typedef struct{
    SPRITE      Sprt;
    BACK_GROUND Bg;
    int         Time;
    int         Point;
    //移動速度
    float     AddX;
    float     AddY;
} EX12_06_STRUCT;

void init12_06(TCB* thisTCB)
{
    EX12_06_STRUCT* work = (EX12_06_STRUCT*)thisTCB->Work;

    //使用するテクスチャの読み込み
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥0055.png",&g_pTex[0] );
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥race_bg.png",&g_pTex[1] );
```

```
    work->Sprt.X = START_X;
    work->Sprt.Y = START_Y;
}

void exec12_06(TCB* thisTCB)
{
    EX12_06_STRUCT* work = (EX12_06_STRUCT*)thisTCB->Work;
    float direction;
    float distance;
    float posX;
    float posY;
//チェックポイントの座標データ
float point_data[][2] =
{//   X      Y
    {512.0,128.0,},   //1箇所目
    {512.0,352.0,},   //2箇所目
    { 96.0,352.0,},   //3箇所目
    { 96.0,128.0,},   //4箇所目
};

    //チェックポイントのチェック
    if(work->Time-- <= 0)
    {//一定時間でチェックポイントを通過
        work->Time = MOVE_TIME;

        //目標ポイントからの相対座標を計算
        posX = point_data[work->Point][0] - work->Sprt.X;
        posY = point_data[work->Point][1] - work->Sprt.Y;

        //目標ポイントへの方向を計算
        direction = atan2( posY, posX);

        //目標ポイントへの距離を計測し、所要時間から速度を割り出す
        distance = sqrtf( (posX * posX) + (posY * posY) );

        work->AddX = cos( direction ) * distance / MOVE_TIME;
        work->AddY = sin( direction ) * distance / MOVE_TIME;

        //次のポイント
        work->Point++;
```

```
        if( work->Point >= POINT_MAX ) work->Point = 0;
    }

    work->Sprt.X += work->AddX;
    work->Sprt.Y += work->AddY;

    //背景とボールの表示
    BGDraw( &work->Bg, 1);
    SpriteDraw( &work->Sprt, 0);
}
```

12-7 逆走の判定

逆走を知る

レースゲームでは、車が逆走をしているかを知りたくなる事があります。

これはプレイヤーが誤ってコースを逆に走った時に注意を促したり、周回数を数える時に逆走を除外したい時などがあるからです。

ではどうやって逆走をしているかを知るのでしょうか？

方法は幾つかありますが、一番現実的なのは、コースデータに、逆走チェック用の区間データを設け、それをチェックしていく事です。

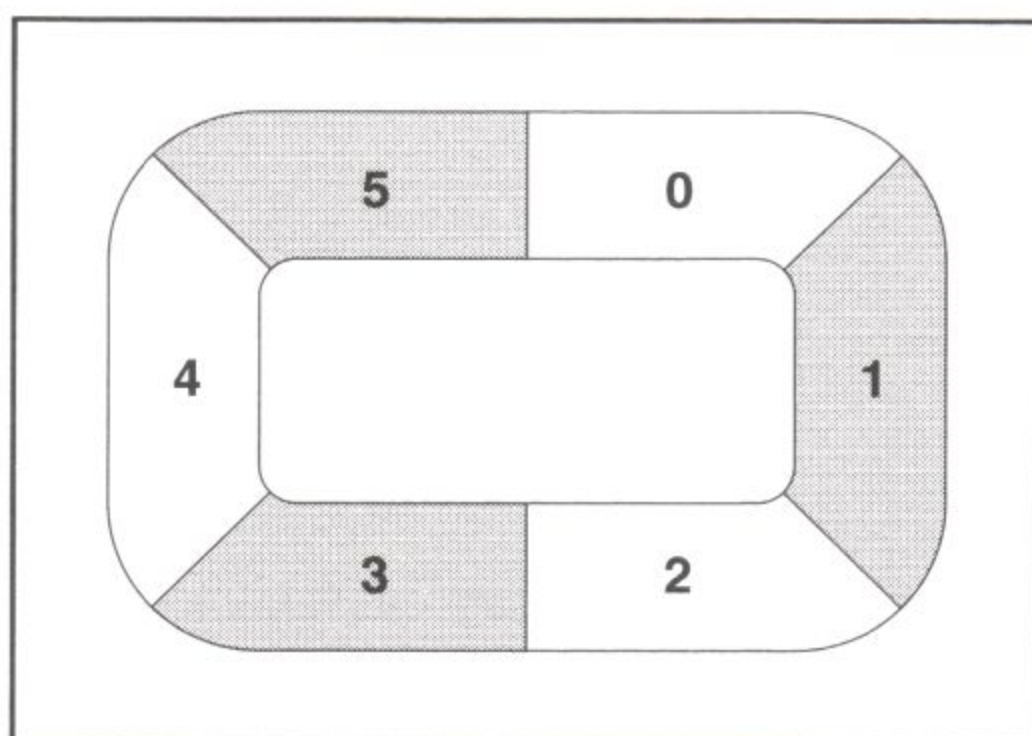
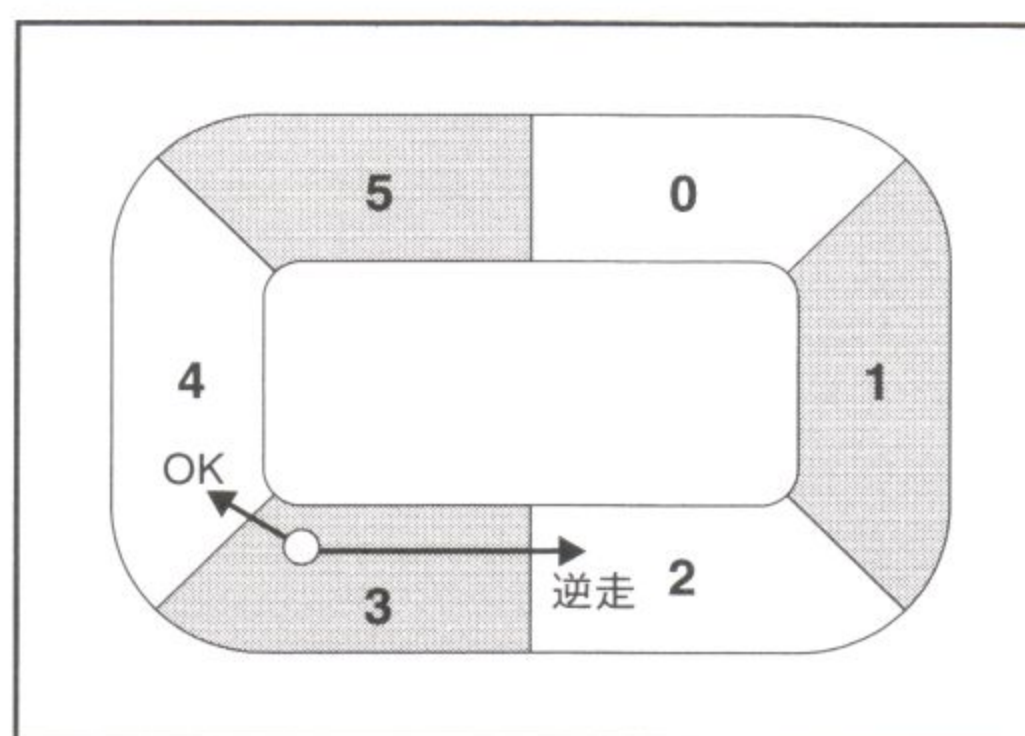
もう少しく詳しく説明します。まず、現在の区間上の位置をチェックします。

そして、移動した区間のデータが本来進む区間のデータでなかった場合、それは本来進むコースではない、すなわち逆走となります。

図12－7－1 逆走チェックのイメージ図

5
0
4
1
3
2

コース上に区間を設ける

5
0
4
1
OK
3
逆走
2

区間をまたがる時に、負の方向へ移動するようなら、
逆走（時計回りの時）

逆走チェックの処理

では実際にプログラムを見ていきましょう。サンプルは方向キーで自由にコース上のボールを動かす物です。この時、反時計回りに移動すると逆走とみなされ、メッセージが表示されます。

はじめにチェック用のデータを定義します。データはそれぞれ区間ごとに0～5の番号が振られています。なお、－1は進入禁止区域を表します。

次に処理です。まず、1フレーム前の座標を保存しておき、移動処理を行ないます。

そして現在の座標データを取得し、マップデータからチェック用のデータを取得します。

この時、もしチェックデータが－1（進入禁止）であれば、座標を先ほど保存しておいた、移動前のデータに戻します。

進行・逆走処理

次に区間のデータをチェックし、新たな区間に進んでいれば、進行の処理を行ないます。次の区間は変数ForwordDataに格納され、現在位置の次の区間を代入します。

次は肝心の逆走の処理です。この処理は単純で、該当区間（次に進行する区間の1つ前の区間）以外の区間に移動したらそれは逆走とみなす事が出来ます。

最後に、逆走の状態をメッセージで表示して、処理は終了です。

LIST 12－7－1 逆走を知る

```
//データ値の最大値
#define DATA_MAX 6
#define MOVE_SPEED 4
#define START_X   320
#define START_Y   96
//マップデータの大きさ(ピクセルサイズ)
#define DATA_SIZE 32
#define DATA_WIDTH 20
#define DATA_HEIGHT 15
#define BALL_CENTER 16

typedef struct{
    SPRITE      Sprt;
    BACK_GROUND Bg;
    char        ForwardData;
    int         ReverseFlag;
} EX12_07_STRUCT;

void init12_07(TCB* thisTCB)
{
```

```
    EX12_07_STRUCT* work = (EX12_07_STRUCT*)thisTCB->Work;

    //使用するテクスチャの読み込み
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥0055.png",&g_pTex[0] );
    D3DXCreateTextureFromFile( g_pD3Ddevice,
"..¥¥..¥¥data¥¥race_bg.png",&g_pTex[1] );

    work->Sprt.X = START_X;
    work->Sprt.Y = START_Y;
}

void exec12_07(TCB* thisTCB)
{
    EX12_07_STRUCT* work = (EX12_07_STRUCT*)thisTCB->Work;
    float old_posX;
    float old_posY;
    char chk_data;
    char reverse_data;
    RECT font_pos = {0,0, 640, 480, };
//区間データ
char map_data[ DATA_HEIGHT ][ DATA_WIDTH ] =
{//0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19
{ -1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1, },  //0
{ -1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1, },  //1
{ -1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1, },  //2
{ -1,-1, 4, 4, 5, 5, 5, 5, 5, 5, 0, 0, 0, 0, 0, 1, 1,-1,-1,-1, },  //3
{ -1,-1, 4, 4, 5, 5, 5, 5, 5, 5, 0, 0, 0, 0, 0, 1, 1,-1,-1,-1, },  //4
{ -1,-1, 4, 4,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1, 1, 1,-1,-1,-1, },  //5
{ -1,-1, 4, 4,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1, 1, 1,-1,-1,-1, },  //6
{ -1,-1, 4, 4,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1, 1, 1,-1,-1,-1, },  //7
{ -1,-1, 4, 4,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1, 1, 1,-1,-1,-1, },  //8
{ -1,-1, 4, 4,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1, 1, 1,-1,-1,-1, },  //9
{ -1,-1, 4, 4, 3, 3, 3, 3, 3, 3, 2, 2, 2, 2, 2, 1, 1,-1,-1,-1, },  //10
{ -1,-1, 4, 4, 3, 3, 3, 3, 3, 3, 2, 2, 2, 2, 2, 1, 1,-1,-1,-1, },  //11
{ -1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1, },  //12
{ -1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1, },  //13
{ -1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1, },  //14
};
```

```
//過去の座標を保持
old_posX = work->Sprt.X;
old_posY = work->Sprt.Y;

//キー入力による移動
if( g_InputBuff & KEY_UP    ) work->Sprt.Y -= MOVE_SPEED;
if( g_InputBuff & KEY_DOWN  ) work->Sprt.Y += MOVE_SPEED;
if( g_InputBuff & KEY_RIGHT ) work->Sprt.X += MOVE_SPEED;
if( g_InputBuff & KEY_LEFT  ) work->Sprt.X -= MOVE_SPEED;


//現在座標のデータを取得
chk_data =
    map_data [ (int)(work->Sprt.Y+BALL_CENTER)/DATA_SIZE ]
             [ (int)(work->Sprt.X+BALL_CENTER)/DATA_SIZE ];

//進入禁止区域かチェックする
if(chk_data == -1)
{//もし進入できなければ、座標を戻す
    work->Sprt.X = old_posX;
    work->Sprt.Y = old_posY;
    //データも再度取得
    chk_data =
        map_data [(int)(work->Sprt.Y+BALL_CENTER)/DATA_SIZE ]
                 [(int)(work->Sprt.X+BALL_CENTER)/DATA_SIZE ];
}

//データタイプを比較し前進しているかをチェック
if(work->ForwardData == chk_data)
{//前進時の処理
     //次に前進したとみなすデータタイプを設定
     work->ForwardData++;
     //データ値が最大なら次のチェックは0
     if(work->ForwardData >= DATA_MAX)
         work->ForwardData = 0;
}

//逆走のチェック
reverse_data = work->ForwardData-1;
```

```
    if(reverse_data < 0) reverse_data = DATA_MAX-1;

    if(chk_data != reverse_data)
    {//もし該当エリア以外に進んだら、逆送とみなす
        work->ReverseFlag = true;
    }else{
     //そうでなければ逆走解除
        work->ReverseFlag = false;
    }

    if(work->ReverseFlag)
    {//逆走中はメッセージを表示
        g_pFont->DrawText(
            NULL,"逆送しています", -1, &font_pos, DT_CENTER, 0xffffffff);
    }
    //背景とボールの表示
    BGDraw( &work->Bg, 1);
    SpriteDraw( &work->Sprt, 0);
}
```

12-8 時間を計るには

ゲームで使用する時間計測

時間の計測は、あちこちで使われます。中でもレースでのラップタイムやシューティングでのタイムアタックは代表的な例でしょう。

この時、フレーム数をカウントする事で時間を計っても良いのですが、処理速度が不足した時など、正確な時間が計れない事があります。

そこでここでは、正確な時間を計る方法を紹介します。

正確な時間を計るには

正確な時間を計るには、WindowsのAPI関数timeGetTimeを使用します。

この関数はPCが起動してからの時間を1/1000秒単位で返す物で、処理速度などに依存せずに時間を返します。

計測プログラム

計測のプログラムはかなりシンプルです。まず、初期化時に計測開始時の時間を記録してやります。

そうしたら、あとは時間を取得したい時に関数timeGetTimeを呼び出し、計測開始時の時間を引いてやります。

これだけで経過時間を1/1000秒単位で得ることが出来ます。

LIST 12 －8－1 時間を計るには

```
typedef struct{
    int   StartTime;
} EX12_08_STRUCT;

void init12_08(TCB* thisTCB)
{
    EX12_08_STRUCT* work = (EX12_08_STRUCT*)thisTCB->Work;

    //計測開始の時間を記録しておく
    work->StartTime = timeGetTime();
```

```
}

void exec12_08(TCB* thisTCB)
{
#define SECOND 1000
    EX12_08_STRUCT* work = (EX12_08_STRUCT*)thisTCB->Work;
    int time;
    char str[128];
    RECT font_pos = {  0,  0,640,480,};

    //開始時間から経過した時間を計算して取得
    time = timeGetTime() - work->StartTime;

    //経過時間の表示
    sprintf( str,"経過時間 %2d.%03d",time / SECOND,time % SECOND);
    g_pFont->DrawText( NULL, str, -1, &font_pos, DT_LEFT, 0xffffffff);
}
```

12-9 10進→16進変換

基数変換

10進数から16進数への変換を行なってみましょう。

こういった、ある表記法から別の表記法を変換することを基数変換といいます。

こういった基数変換はゲームで使われる場面はさほど多くありませんが、テキストファイルからの入力や、ユーザーが直接入力した文字等、まれに使用する事がありますので覚えておいて損はありません。

変換プログラム

さて、実際のプログラムですが、まず変換したい数値で、実際に使用されている桁数をカウントします。

次に、各桁に該当した値を各桁の基数倍にして、合計値に足しこんでやります。

これは例えば、2桁目の数値が3だった場合、2桁目の基数は10なので、3×10で30にして足しこむ、という事です。

これを各桁に対して行なえば、変換が行なえます。

なおこの際、上位桁からではなく、最下位の桁から計算していくと、基数を計算する時に乗算だけですむので処理が楽になります。

LIST 12－9－1 10進→16進変換

```
void exec12_09(TCB* thisTCB)
{
//10進数からの変換
#define BASE_RADIX 10
    char str[128];
    RECT font_pos = {  0,  0,640,480,};

    int ans = 0;
    int digit = 1;
    int digit_count = 0;
    //変換用の10進数データ
```

```
    char* in_str = "32579";


    //1桁目(右端)を検出
    while( in_str[ ++digit_count ] != NULL );

    for(digit_count--; digit_count >= 0; digit_count--)
    {//各桁ごとに基数の桁を掛けてやる
        ans += (in_str[ digit_count ] - '0') * digit;
        //下位桁から上位桁数値を算出
        digit *= BASE_RADIX;
    }

    //変換数値の表示
    sprintf( str,"変換数値 %d",ans);
    g_pFont->DrawText( NULL, str, -1, &font_pos, DT_LEFT, 0xffffffff);
}
```

12-10 16進→10進変換

応用のきく変換プログラム

16進数から10進数への変換を行なってみましょう。

通常扱っている数値を10進数へ変換するだけなら、sprintf等の変換関数を使えば良いのですが、ここではそれ以外の、たとえば12進数や24進数などでも使える手法を紹介します。

変換の方法

まず、始めに変換可能な桁数を決定します。BorlandC++で扱える型は32BITなので扱える数値は0～4294967295となり、変換可能な桁数は（10進数で）10桁になります。

次に、その変換先の進数の該当桁に対応する変換テーブルを桁数ー1個分用意します。

ここでは10進数に変換するため、1,000,000,0000～10の各桁のテーブルを用意します。

なお、最後の1つは0になるため必要ありません。

```
//除算に使用する各桁のテーブル
    int  div_tbl[] =
    { 1000000000,
       100000000,
        10000000,
         1000000,
          100000,
           10000,
            1000,
             100,
              10,
    };
```

実際のアルゴリズムは単純で、指定した桁数を上位桁から順番に除算してゆき、商をその桁数とします。この際、次の桁を計算する時に、その余りを被除数とします。

これを、桁数分繰り返してやれば変換終了となります。

サンプルでは、除算の結果を文字列に直接変換しているため、最後に終端文字列を追加しています。

LIST 12 －10－1 16進→10進変換

```
void exec12_10(TCB* thisTCB)
{
    char str[128];
    RECT font_pos = {  0,  0,640,480,};

    int loop;
    int in_num = 1024;
    char str_buff[11];
    char cnv_tbl[10] = "0123456789";
    int  div_tbl[] =
{ 1000000000,
   100000000,
    10000000,
     1000000,
      100000,
       10000,
        1000,
         100,
          10,
};
//16進→10進変換
    for( loop=0; loop < 9; loop++ )
    {
        //1桁毎に文字列へ変換
        str_buff[ loop ] = cnv_tbl[ in_num / div_tbl[ loop ] ];
        if( str_buff[ loop ] != '0' )
        {//各桁ごとに商を引く(余りを求める)
            in_num %= div_tbl[ loop ];
        }
    }
    str_buff[ loop ] = cnv_tbl[ in_num ];
    //終端文字列を加える
    str_buff[ loop+1 ] = NULL;
//変換数値の表示
    sprintf( str,"変換数値 %s",str_buff);
    g_pFont->DrawText( NULL, str, -1, &font_pos, DT_LEFT, 0xffffffff);
}
```

12-11 ラジアン⇔度数変換

ラジアンと度数

角度の表記法であるラジアンと度数の相互変換を行なってみましょう。

プログラムで扱う場合はラジアンの方が便利な事が多く、また多くの数学関数がラジアンを基準に関数を設計していますので、通常使用の際はあまり度数を使う事はありません。

しかし、プレイヤーに角度の情報を知らせる時に、ラジアンで伝えるというのでは、非常に解りにくく、またプログラマでも、度数法で考えた方が理解しやすい局面は多々あります。

そこで、ラジアン、度数を相互に変換する必要が出てきます。

変換プログラム

実際のプログラムですが、ラジアンと度数の関係式をそのままプログラムしています。

図12－11－1 ラジアン、度数の相互変換式の図

度数からラジアンへ
ラジアン ＝ 度数 ÷ 180 × π

ラジアンから度数へ
度数 ＝ ラジアン ÷ π × 180

特に注意する所はありませんが、1つだけある定数、M_PIは浮動小数点定数で、πを表しています。

```
//ラジアン⇔度数変換
float degree2rad( float degree )
{
    return degree / 180.0 * M_PI;
}

float rad2degree( float rad )
{
    return  rad / M_PI * 180.0;
}
```

Chapter 13

TIPS

逆引きゲームプログラミング

Game programming

13-1 よく使う数学関数解説 sin

sin関数

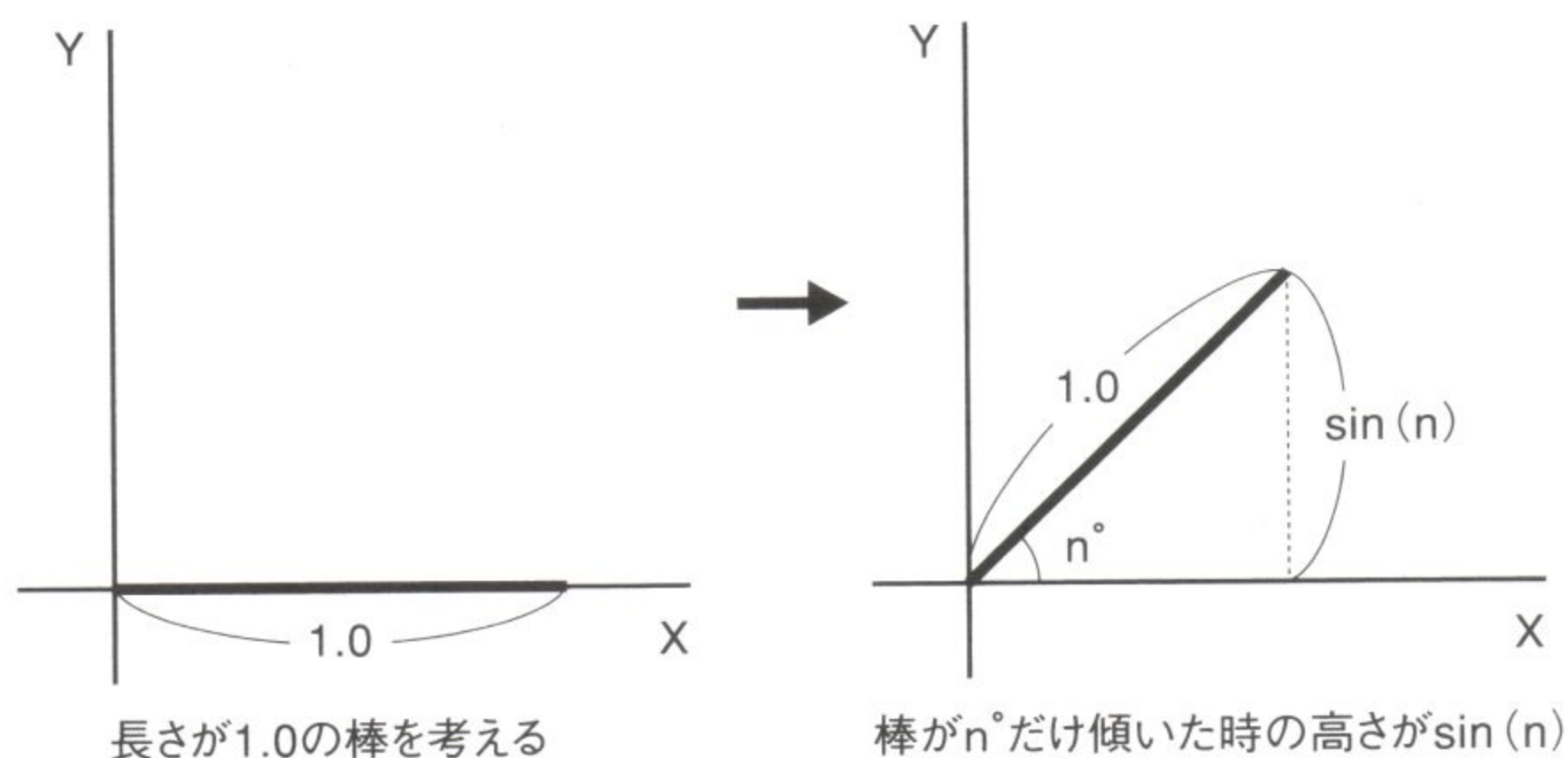
sin関数とは、「正接」を求める関数です。

とはいっても、これでは分からないと思いますので、もう少し噛み砕いて言うと「角度がn度の時のYの長さを返す関数」です。

これは正確な言い方ではありませんが、sin関数の性質を表しています。

図13－1－1 sin関数の解説図

長さが1.0の棒を考える

棒がn°だけ傾いた時の高さがsin (n)

具体的な例

まだわかりにくいと思いますので、もう少し詳しく解説します。

まず、1.0の長さを持つ棒を想像してみてください。そしてこれをX軸に対して平行に置きます。

この時、Xの長さは1.0、Yの長さ(高さ)は0.0になります。

そしてこれをn度だけ傾けると、当然Yの長さが増えます。この長さを返す関数が、すなわちsin関数なのです。

この時返ってくる値は、1.0の長さの「棒」を渡した時とされ固定です。

任意の長さの棒の時は帰ってきた値に、好きな倍率をかけてやると得る事が出来ます。

もちろん、これだけでsin関数の性質を完全に説明した訳ではありませんが、このイメージを取っ掛かりにすると理解が早いかと思います。

なお、関数に渡す角度は、一般に度数単位ではなくラジアン単位ですので注意してください。

13-2 よく使う数学関数解説 cos

cos 関数

sin 関数が Y の長さを返すならば、X の長さを返す関数も当然存在します。

想像がつくと思いますが、それが cos 関数です。

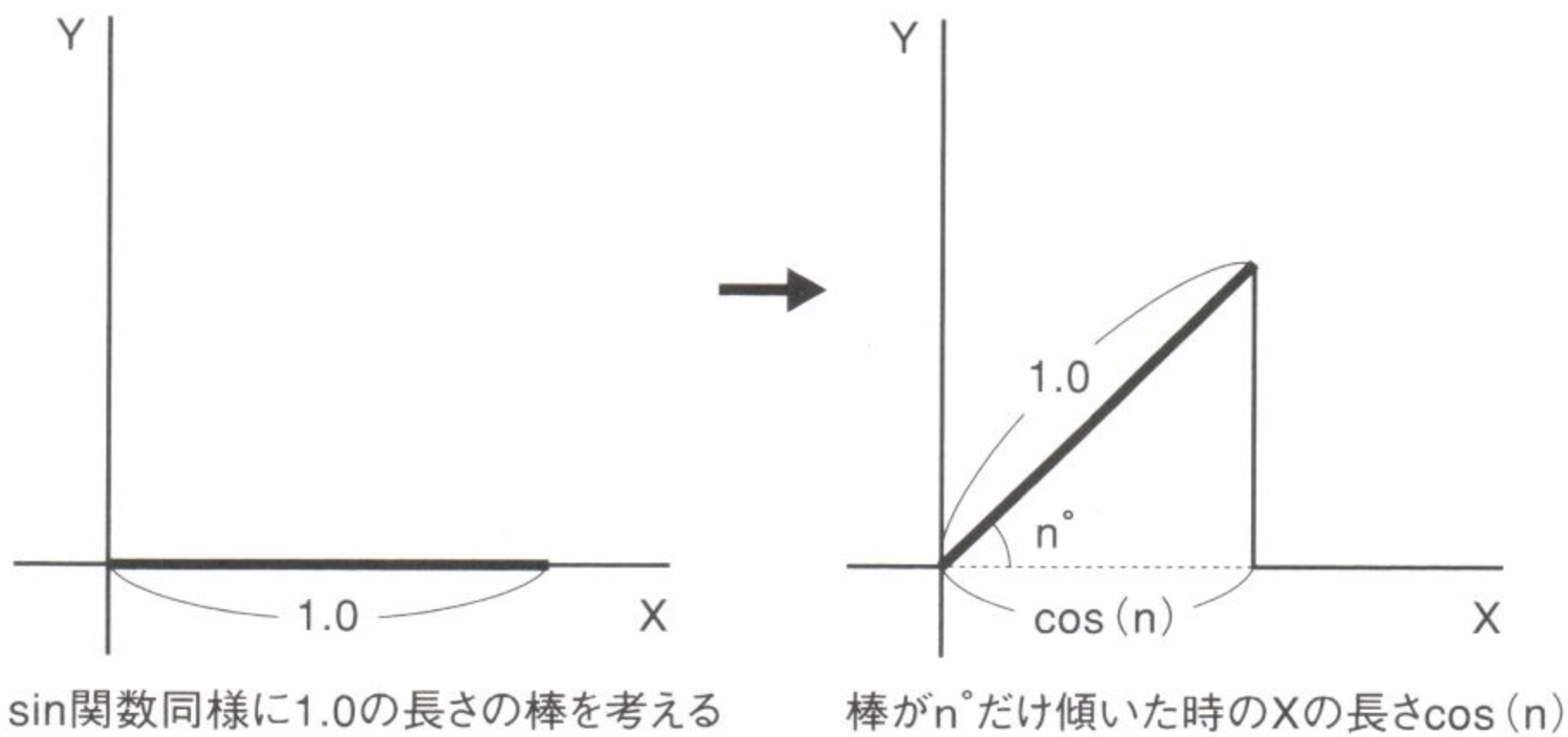
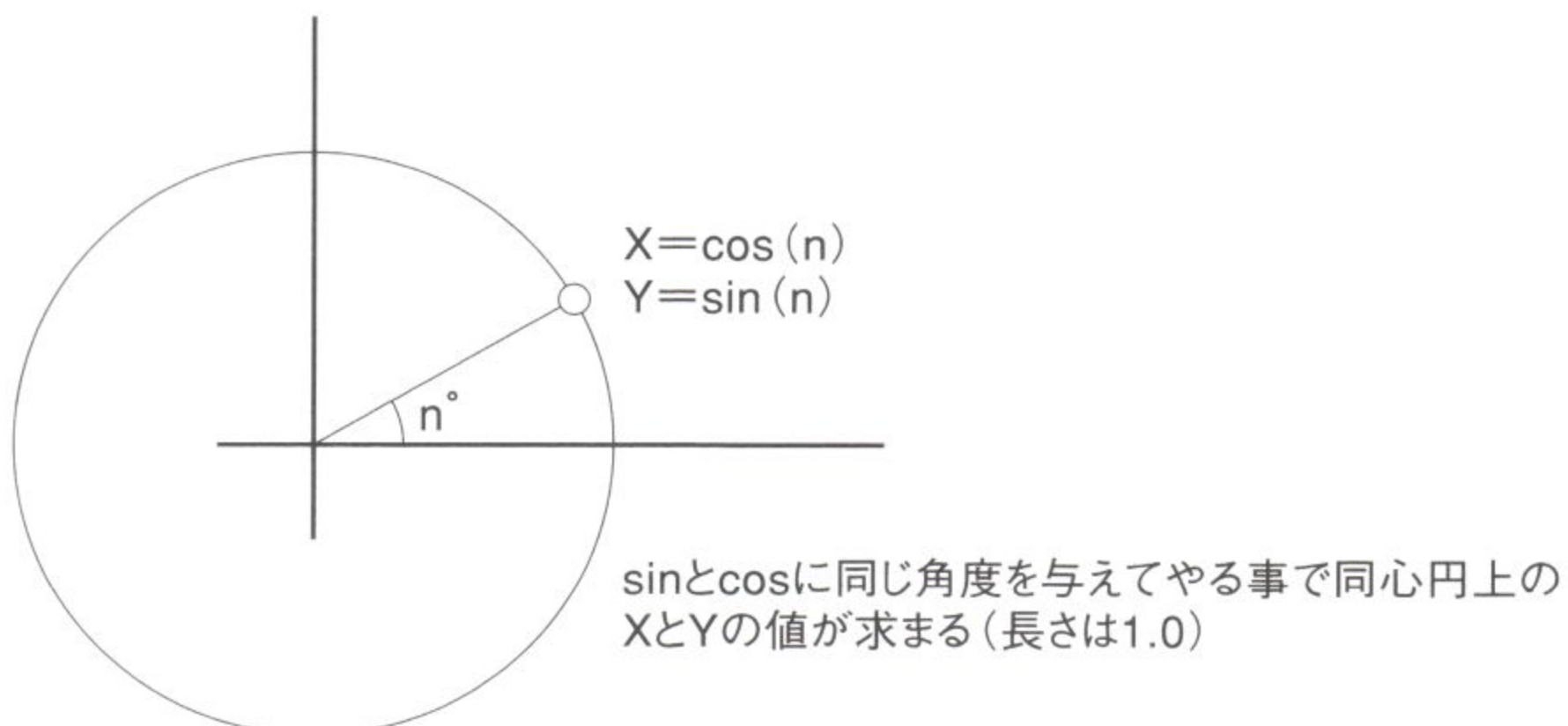
この関数は、sin 関数と対になる性質をもち、「角度が n 度の時の X の長さを返す」関数です。

この cos 関数と、sin 関数に同じ値（角度）を渡す事で、同一の長さをもつ点の座標、すなわち円の座標を得る事ができます。

図 13－2－1 cos 関数の解説図

sin関数同様に1.0の長さの棒を考える　　棒がn°だけ傾いた時のXの長さcos (n)

図 13－2－2 円の座標の解説図

sinとcosに同じ角度を与えてやる事で同心円上のXとYの値が求まる（長さは1.0）

また、cos 関数は sin 関数を 90 度ずらした関数である事が知られています。

この性質を利用すると、sin 関数だけで円を描く事も出来ます。

13-3 よく使う数学関数解説 atan

atan関数

atan関数とは、一言で言うと「XとYの比率から、角度を返す」関数の事です。

帰ってくる角度はラジアン単位で$-\pi \sim \pi$の値を返します。

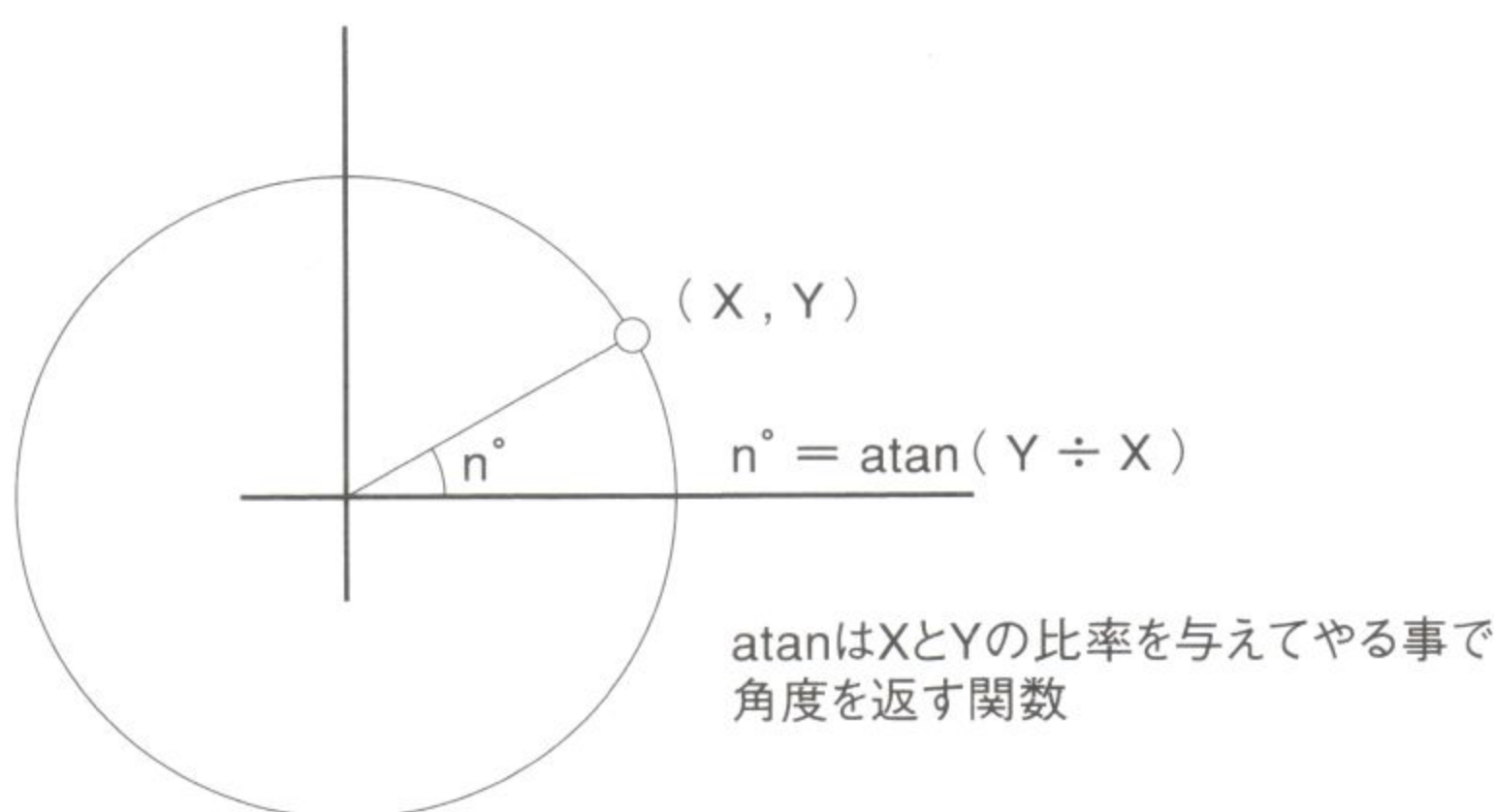
この関数は、XとYの座標から角度を求めたい時に良く使われます。

しかし、実際にatan関数を使用してみると分かりますが、0除算を考慮しなくてはならない等、少々使いにくい所があります。

そのため、実際にはatan関数の代わりにatan2関数の方が良く使用されます。

これは引数に比率を渡すのではなく、XとYを別々に渡すようにした関数で、使う際に不都合な点が考慮されています。

通常はこちらを使用するようにすると良いでしょう。

図13−3−1 atan関数の解説図

この関数はゲームでは非常に有用な関数で、相手の座標から方向を得るなど応用範囲も広く、事実上必須といっても良い関数です。

そのため、使用方法は確実に理解するようにしておいたほうが良いでしょう。

13-4 よく使う数学関数解説 sqrt

sqrt関数

sqrt関数は平方根を求める関数です。

もう少し平たく書くと、「２乗してｎになる数」を求める関数です。

sqrt関数の使用例

これだけでは何の意味があるのか良く分から無いと思いますが、この関数はゲームでは２点間の距離を求める関数として頻繁に使用されます。

使用方法はさほど難しくなく、A、B、２点間の距離を求めたい場合に計算式

$\sqrt{(AX - BX)^2 + (AY - BY)^2}$ を適用します。

図13－4－1 2点間の距離

2点の間の距離は公式で求められる

$\sqrt{(AX - BX)^2 + (AY - BY)^2}$ ＝距離

プログラムで書くと、以下のような形になります。

```
X = A_X - B_X;
Y = A_Y - B_Y;
distance = sqrtf( X * X + Y * Y );
```

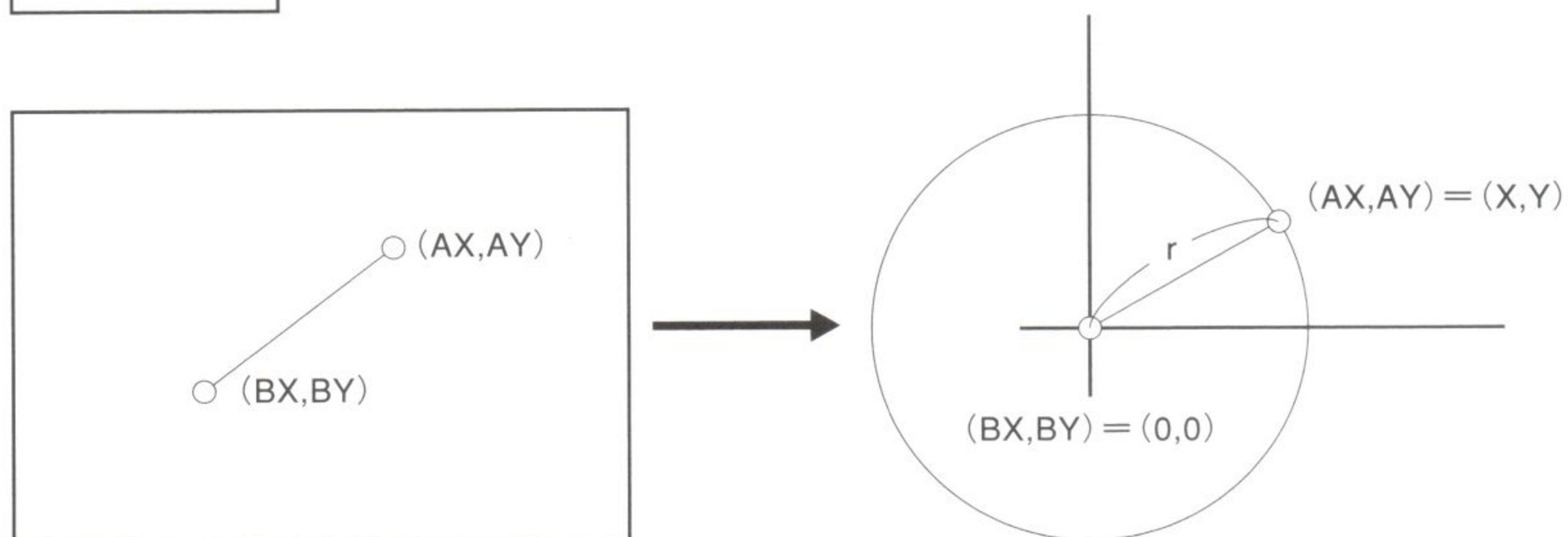
この式がピンとこない方もいると思いますので、少し詳しく解説します。

これは $x^2 + y^2 = r^2$（rは半径）という、円の公式を応用した物です。

これは要するに「Xの２乗とYの２乗を足した物は円の半径の２乗に等しい」という事を言っています。

半径の２乗の平方根は当然半径ですから、この事はすなわち「Xの２乗とYの２乗を足した物の平方根は半径である」という事を意味しています。

図 13 －4－2 円の公式

$X^2 + Y^2 = r^2$
円の公式

なお、この公式の応用は円だけではなく3次元の球にも当てはまります※。
そのため、3次元での距離の計測や、当たり判定などにも頻繁に使用されます。

※ $x^2 + y^2 + z^2 = r^2$ となるため、XYZそれぞれの2乗を加算した値の平方根を求める事で球の半径が求まる。

Index

索引

逆引き ゲームプログラミング
Game Programming

Index 索引

た

な

は

ま

や

ら

わ

万里　ゆうじ（まり　ゆうじ）

某アーケードゲームメーカーに就職し、格闘ゲームやアクションゲームを製作する。
その後、転職し、PSやPS2のゲーム製作にも関わる。
また、プログラム以外に企画も行なっており、マルチに仕事をこなす。
（只の器用貧乏という話もあり）

逆引き ゲームプログラミング

発行日　2005年11月11日　第1版第1刷
　　　　2007年 5月 1日　第1版第4刷

著　者　万里　ゆうじ

発行者　斉藤　和邦
発行所　株式会社　秀和システム
　　　　〒107-0062　東京都港区南青山1-26-1 寿光ビル5F
　　　　Tel 03-3470-4947（販売）
　　　　Fax 03-3405-7538
印刷所　株式会社シナノ

ISBN4-7980-1169-X C3055

◆サンプルファイルのトラブル

◆症状 1　コンパイルできない
コンパイル時に「"make"は内部コマンド、または外部コマンド、操作可能なプログラムまたはバッチファイルとして認識されていません」等と表示される場合。

◇推測される原因　BCCのBinフォルダへパスが通っていない
→対策：
現状でのBCCのBinフォルダにパスを通してください。
あるいは、BCCを書籍でインストールしている場所と同じところにインストールしなおしてください。

◆症状 2　コンパイルできても、生成されたサンプルファイルが実行出来ない
「序数14がダイナミックライブラリD3D9.DLLからみつかりませんでした。」など表示される場合。

◇推測される原因　SDKのバージョンが古い
→対策：
DirectX SDKのバージョンを最新にして、コンパイルしなおしてください。

→Windows2000などで、最新のSDKが利用出来ない場合：
SDKがDecember2004などのバージョンだとコンパイルできても、サンプルが動作しない可能性があります。
その場合は下記の方法で、最新版のDirectXランタイムのDLLからライブラリを構築してみてください。

1. DirectXのエンドユーザー向けランタイムの最新版（2006/1/17現在December2005）をインストール
2. WinNTフォルダ中から、1-4(6ページ)に記載されている6個のDLL（全てない場合は、とりえず存在するファイル全部）を、¥DEVELOP¥MAKE_LIB¥DLLにコピー
3. ¥DEVELOP¥MAKE_LIBのmake.batを実行（LIBに変換、BCCの指定フォルダに格納される）
4. この状態で再びコンパイル

これでも動作しない場合、ランタイムと同じバージョンのSDK（上記の場合December2005）の中から、DLLを抜き出してLIBに変換するなど、いくつか試してみてください。

●176ページ　「多数のキャラクターを表示する」のソースコード　下から5行目
【誤】→　for(i=0; i>3; i++){
【正】→　for(i=0; i<3; i++){

●274ページ　LIST6-12-1「振り子のような動き」のソースコード　14行目
【誤】→　#define PENDULUM_RADIUS 1.0　　　　//振り子の往復の幅
【正】→　#define PENDULUM_RADIUS2 1.0　　　　//振り子の往復の幅

●274ページ　LIST6-12-1「振り子のような動き」のソースコード　下から4～5行目
【誤】→
```
pspr->X = PENDULUM_CENTER_X + sin( sin( pspr->Count) * PENDULUM_RADIUS ) * PENDULUM_RADIUS;
pspr->Y = PENDULUM_CENTER_Y + cos( sin( pspr->Count) * PENDULUM_RADIUS ) * PENDULUM_RADIUS;
```
【正】→
```
pspr->X = PENDULUM_CENTER_X + sin( sin( pspr->Count) * PENDULUM_RADIUS2 ) * PENDULUM_RADIUS;
pspr->Y = PENDULUM_CENTER_Y + cos( sin( pspr->Count) * PENDULUM_RADIUS2 ) * PENDULUM_RADIUS;
```

●322ページ　LIST7-1-1「キャラのアニメーション」のソースコード　7～10行目
【誤】→
```
{ {  0, 0, 16, 32,},        //1枚目のアニメパターン
  { 16, 0, 32, 32,},        //2枚目のアニメパターン
  { 32, 0, 48, 32,},        //3枚目のアニメパターン
  { 48, 0, 64, 32,},        //4枚目のアニメパターン
```
【正】→
```
{ {  0, 0, 64,128,},        //1枚目のアニメパターン
  { 64, 0,128,128,},        //2枚目のアニメパターン
  {128, 0,192,128,},        //3枚目のアニメパターン
  {192, 0,256,128,},        //4枚目のアニメパターン
```

●586ページ　図12-11-1
【誤】→

度数からラジアンへ	ラジアンから度数へ
度数=ラジアン÷180×π	ラジアン=度数÷π×180

【正】→

度数からラジアンへ	ラジアンから度数へ
ラジアン=度数÷180×π	度数=ラジアン÷π×180